TravelMatch

Software Requirements Document

Version 1.0

D.J. van den Brand (0772180)
S. He (0810831)
J.M.A.P. van Hoof (0778486)
G.C. Linders (0815449)
M.J.M. Muijsers (0805654)
G.H. Santegoeds (0890429)
L.D. Stooker (0819041)
J.W.J.H. Visser (0828234)

22nd June, 2015

Abstract

This document contains the software requirements for the TravelMatch application, which is developed as part of the Software Engineering Project at Eindhoven University of Technology. The requirements in this Software Requirements Document satisfy the requirements in the User Requirements Document. [3] This document complies with the Software Engineering Standard, as specified by the European Space Agency. [1]

Contents

Document Status Sheet **3**
General . . . 3
Document history . . . 3

Document Change Records **4**
General . . . 4
Changes . . . 4

1 Introduction **5**
1.1 Purpose . . . 5
1.2 Scope . . . 5
1.3 Definitions and abbreviations . . . 5
1.3.1 Definitions . . . 5
1.3.2 Abbreviations . . . 6
1.4 References . . . 7
1.5 Overview . . . 8

2 General description **9**
2.1 Relation to current projects . . . 9
2.2 Relation to predecessor and successor projects . . . 9
2.3 Function and purpose . . . 9
2.4 Environment . . . 9
2.5 Relation to other systems . . . 10
2.5.1 Affiliate Network . . . 10
2.5.2 Authentication providers . . . 10
2.5.3 Analytics . . . 10
2.6 General constraints . . . 11
2.6.1 Security . . . 11
2.6.2 Image sizes . . . 11
2.6.3 Extendibility . . . 11
2.7 Model description . . . 11
2.7.1 Domain model . . . 11
2.7.2 Model-View-Controller model . . . 13
2.7.3 Data model . . . 14
2.7.4 Class Diagrams . . . 18
2.7.5 Message sequence diagrams . . . 24

3 Specific requirements **45**
3.1 Functional requirements . . . 45
3.1.1 Description requirements . . . 45
3.1.2 User . . . 46
3.1.3 Vacation . . . 52
3.1.4 CMS . . . 65
3.1.5 AI module . . . 67
3.1.6 Analytics . . . 69
3.1.7 Affiliate network . . . 69
3.2 Non-Functional requirements . . . 70
3.2.1 Content Management System . . . 70

3.2.2 API . . . 71
3.2.3 TravelMatch app . . . 71
3.2.4 Analytics . . . 73
3.2.5 Adaptability . . . 73
3.2.6 Capability . . . 73
3.2.7 Performance . . . 73
3.2.8 Licensing . . . 74

4 Requirements traceability matrix **75**
4.1 User requirements to software requirements . . . 75
4.2 Software requirements to user requirements . . . 80

A User Interface **88**
A.1 User application . . . 88
A.1.1 Splash screen . . . 89
A.1.2 Start screen . . . 90
A.1.3 Vacation details . . . 91
A.1.4 Sidebar . . . 92
A.1.5 Registration . . . 93
A.1.6 Registration error . . . 94
A.1.7 Logging in . . . 95
A.1.8 Profile . . . 96
A.1.9 Interest analysis example . . . 97
A.1.10 Loading screen . . . 98
A.1.11 Hotel overview screen . . . 99
A.1.12 Hotel details . . . 100
A.2 CMS . . . 101
A.2.1 Login . . . 101
A.2.2 Overview . . . 102
A.2.3 Adding images . . . 103
A.2.4 Images overview . . . 103
A.2.5 User overview . . . 104
A.2.6 Single user . . . 104

Document Status Sheet

General

Document title:	Software Requirements Document	
Document identifier:	TravelMatch.Doc.SRD/1.0	
Authors:	D.J. van den Brand	(0772180)
	S. He	(0810831)
	J.M.A.P. van Hoof	(0778486)
	G.C. Linders	(0815449)
	M.J.M. Muijsers	(0805654)
	G.H. Santegoeds	(0890429)
	L.D. Stooker	(0819041)
	J.W.J.H. Visser	(0828234)
Document status:	Final document	

Document history

Version	Date	Author(s)	Reason
0.1	20-05-2015	G.C. Linders, G.H. Santegoeds	Initial version.
0.2	22-05-2015	G.C. Linders, G.H. Santegoeds, D.J. van den Brand.	First requirements.
0.3	28-05-2015	G.C. Linders, L.D. Stooker, J.W.J.H. Visser, G.H. Santegoeds	Second draft.
0.4	01-06-2015	L.D. Stooker, J.W.J.H. Visser, G.H. Santegoeds	Third draft.
0.5	03-06-2015	L.D. Stooker, J.W.J.H. Visser, G.H. Santegoeds, D.J. van den Brand	Fourth draft.
0.6	05-06-2015	L.D. Stooker	Fifth draft.
0.7	09-06-2015	L.D. Stooker	Sixth draft.
0.8	12-06-2015	L.D. Stooker	Seventh draft.
1.0	17-06-2015	L.D. Stooker	Final version.

Document Change Records

General

Date:	22nd June, 2015
Document title:	Software Requirements Document
Document identifier:	TravelMatch.Doc.SRD/1.0

Changes

Section	Reason

Chapter 1
Introduction

1.1 Purpose

This Software Requirements Document (SRD) provides a translation from the specific user requirements as listed in Chapter 3 of the User Requirements Document (URD)[3] to the software requirements required for the TravelMatch application. Whereas the user requirements state the requirements of the software as specified by the client, these software requirements state the requirements of the software that the developers place upon the system in order to fulfill the user requirements. Specifically, the software requirements dictate, in developer terms, *what* TravelMatch must do, and not *how* – in other words, it is implementation-independent. These requirements are modeled in a logical model, which provides a simplified view of the systems content and behavior.

1.2 Scope

TravelMatch is an application designed for smartphones and tablets, conceived by iLysian B.V. and developed by the TravelMatch development team. The purpose of the application is to assist users in planning a vacation by showing them images from various destinations and hotels or other places to stay. The application employs machine learning to build a profile of the user in order to suggest the ideal trip.

1.3 Definitions and abbreviations

1.3.1 Definitions

Affiliate Network	A network that enables you to receive money from customer redirection [20]
Analytics Data	The log of analytics events that is recorded and stored on the database.
Android	A popular open-source operating system for embedded devices, including smartphones and tablets, created by Google.
Angular JS	An open-source web application framework maintained by Google.
Cosine similarity	A measure of similarity between two vectors of an inner product space that measures the cosine of the angle between them.
Destination advice	The city, and selection of hotels, that is advised to a user after performing one or more interest analyses.
Destination attributes or tags	Each destination will have one or more *destination attributes* with an associated numerical relative value, those attributes cover the same preferences as the *DNA attribute*.
DNA attribute or tags	These are the attributes that the client wants to use to compose the DNA of. In the beginning 10 attributes are chosen and each image shall have a relative numerical value on one or more of the attributes. Attributes can be added or removed later for new and existing images and DNA.
Google Play Store	A public repository of free and paid apps for Android, managed by Google.

Guest user	An user that does not provide login details but still uses the TravelMatch app.
Hotelstars rating	A hotel classification with common criteria and procedures in participating countries to rate a hotel's quality. See [23].
iLysian	Short for iLysian B.V., a software engineering company situated in Eindhoven, Netherlands. The client for the TravelMatch project.
Interest analysis	The action the user will do of judging the images.
iOS	A popular closed-source operating system for smartphones and tablets created by Apple.
iOS App Store	A public repository of free and paid apps for iOS, managed by Apple.
JWT	JSON Web Token: a compact URL-safe means of representing claims to be transferred between two parties, and used in TravelMatch as authentication token, since it is self-validating.
Relational database management system (RDBMS)	A database management system (a piece of computer software that interacts with users, other applications and a database to capture and analyze data) based on the relational model (commonly based on the relational database model)
TCP/IP	A computer networking model and set of communication protocols used on the internet and similar computer networks, including the Transmission Control Protocol (TCP) and the Internet Protocol (IP)
Tinder	A popular dating application for smartphones and tablets featuring a swipe based interface, where a swipe to the left indicates a dislike and a swipe to the right indicates a like.
Travel DNA	A collection of information about vacation preferences of a specific user or, more specifically, one vacation of that user. This information is stored on the server in a table with values representing the respective gain per attribute for each image the user has swiped.
TravelMatch	An application for smartphones and tablets that assists users in planning a vacation. The subject of this project.
TravelMatch team	A team of Computer Science students at Eindhoven University of Technology who will design and implement the TravelMatch application.
User	The user of the app.
Waverunner	Waverunner Search Service by Pyton Communication Services; a search service that provides vacation offers and prices of participating travel agencies.

1.3.2 Abbreviations

ADT	Abstract Data Type
AI	Artificial Intelligence
APK	Android Application Package
App	Application for smartphones and tablets
CMS	Content Management System
ESA	European Space Agency
GUI	Graphical User Interface
ID	Identifier
IPA	iOS App Store Package
JPEG	Joint Photographic Experts Group; a type of image file
OS	Operating System

SRD	Software Requirements Document
TU/e	Eindhoven University of Technology
URD	User Requirements Document
URL	Uniform Resource Locator
XML	Extensible Markup Language

1.4 References

[1] ESA PSS-05-0 Issue 2, Software requirements and architecture engineering process, February 1991

[2] TravelMatch team. User Requirement Document. Version 1.2.1. 22 June 2015.

[3] TravelMatch team. Software Requirements Document. Version 1.0. 22 June 2015.

[4] TravelMatch team. Architectural Design Document. Version 1.0. 22 June 2015.

[5] TravelMatch team. Detailed Design Document. Version 1.0. 22 June 2015.

[6] TravelMatch team. Software User Manual. Version 1.0. 22 June 2015.

[7] TravelMatch team. Software Transfer Document. Version 1.0. 22 June 2015.

[8] TravelMatch team. Unit Test Plan. Version 1.0. 22 June 2015.

[9] TravelMatch team. Integration Test Plan. Version 1.0. 22 June 2015.

[10] TravelMatch team. Acceptance Test Plan. Version 1.0.2. 22 June 2015.

[11] TravelMatch team. Software Configuration Management Plan. Version 1.0. 22 June 2015.

[12] TravelMatch team. Software Project Management Plan. Version 1.0. 22 June 2015.

[13] TravelMatch team. Software Quality Assurance Plan. Version 1.0. 22 June 2015.

[14] TravelMatch team. Software Verification and Validation Plan. Version 1.0. 22 June 2015.

[15] Tom Preston-Werner. Semantic Versioning 2.0.0. Retrieved 6 May 2015. http://www.semver.org/

[16] Coley Consulting. MoSCoW Prioritisation. Retrieved 29 April 2015. http://www.coleyconsulting.co.uk/moscow.htm

[17] Tinder, Inc. Tinder. Retrieved 29 April 2015. http://www.gotinder.com/

[18] Organized Shopping, LLC. Affiliate Network. Marketing Terms. Retrieved 29 April 2015. http://www.marketingterms.com/dictionary/affiliate_network/

[19] Daiycon. About Daisycon. Retrieved 29 April 2015. http://www.daisycon.com/en/about_daisycon/

[20] Drifty Co. Ionic: Advanced HTML5 Hybrid Mobile App Framework. Retrieved 30 April 2015. http://ionicframework.com/

[21] Hotelstars Union. Classification criteria 2015-2020. Retrieved 1 May 2015. http://www.hotelstars.eu/index.php?id=criteria

[22] Django. http://www.django-cms.org/en/

[23] Django administration module. The Django Django admin site. Retrieved 1 June 2015. https://docs.djangoproject.com/en/1.8/ref/contrib/admin/

[24] Django Software Foundation. The Web framework for perfectionists with deadlines — Django. Retrieved 1 June 2015. https://www.djangoproject.com/

[25] Facebook User ID. User IDs and Friends. Retrieved 2 June 2015. https://developers.facebook.com/docs/apps/upgrading#upgrading_v2_0_user_ids

[26] ImageMagick. ImageMagick: Convert, Edit, Or Compose Bitmap Images. Retrieved 2 June 2015. http://www.imagemagick.org/

[27] Google. AngularJS - Superheroic JavaScript MVW Framework. Retrieved 1 June 2015. https://angularjs.org

[28] Adobe Systems Inc. Phonegap: Home. Retrieved 1 June 2015. http://phonegap.com/

[29] Xamarin Inc. Mobile App Development & App Creation Software - Xamarin. Retrieved 1 June 2015. http://xamarin.com/

[30] Eric Raymond. The Jargon File. Version 4.4.7. Retrieved 17 June 2015. http://www.catb.org/jargon/html/

[31] Python Software Foundation. Classes. The Python Tutorial. Retrieved 18 June 2015. https://docs.python.org/2/tutorial/classes.html

[32] Python Software Foundation. PEP 0008 – Style Guide for Python Code. 1 August 2013. https://www.python.org/dev/peps/pep-0008/

[33] Django Software Foundation. Coding style. Retrieved 18 June 2015. https://docs.djangoproject.com/en/1.8/internals/contributing/writing-code/coding-style/

[34] Django Software Foundation. Writing your first Django app, part 1. Database setup. Retrieved 18 June 2015. https://docs.djangoproject.com/en/1.8/intro/tutorial01/#database-setup

[35] Massachusetts Institute of Technology. MIT License. Retrieved 21 June 2015. http://opensource.org/licenses/MIT

[36] Apache Software Foundation. Apache License, Version 2.0. January 2004. http://www.apache.org/licenses/LICENSE-2.0

1.5 Overview

The remainder of this document consists of three chapters. In chapter 2 we give a general description of the TravelMatch application, including its relation to other projects, its environment and a description of the logical model it uses. Sections 2.1 and 2.2 discuss the relation to current projects, predecessor projects and successor projects. Section 2.3 describes the general function and purpose that the TravelMatch application fulfills. Section 2.4 describes the environment in which TravelMatch will operate. Section 2.5 discusses the relation of the TravelMatch app to other systems. Section 2.6 discusses the general constraints that the TravelMatch team must comply with. Section 2.7 gives a description of the logical model of the TravelMatch app.

Afterwards, in chapter 3 of this document, we list the functional and non-functional requirements that the TravelMatch team must comply with. Chapter 4 contains a traceability matrix that maps the requirements in the URD [3] to the requirements in this SRD.

Chapter 2
General description

2.1 Relation to current projects

The TravelMatch app is related to the project Tinder, which is a dating app that pairs users together if they indicate that they "like" each other's pictures. Tinder uses a swiping interface, where a swipe to the right means a "like" and a swipe to the left means a "dislike". The interface of the TravelMatch app was directly inspired by this swiping interface featured in the Tinder app. Besides the Tinder app, there are no other projects related to TravelMatch.

2.2 Relation to predecessor and successor projects

Travelmatch does not build upon any predecessor project.

After the conclusion of this Software Engineering Project, further development of the TravelMatch app will be handled by iLysian. iLysian may continue to change, add or remove functionality for the TravelMatch app and back end server. The TravelMatch back end server will run on servers provided by iLysian. Furthermore, iLysian will be responsible for maintenance.

2.3 Function and purpose

The TravelMatch app will provide travelers with a new and innovative way to plan their vacation. Users may input their vacation details and are presented with a series of images; each image relates to some aspect of a vacation. We refer to such an aspect as a *DNA attribute*. The user can choose to "like" or "dislike" each image. Based on these choices, TravelMatch will build a *Travel DNA* of the user. This *Travel DNA* is then used by an algorithm to present a recommended destination to the user. The user receives information about available hotels at this destination, specifically tailored to the user's preferences. If the user would like to book the hotel, they may go to the website of a travel agency in order to do so. This is done via an affiliate network, namely TradeTracker. In the future, a direct connection may be added to facilitate in-app booking. However, this functionality falls outside the scope of this project.

The purpose of the TravelMatch app is to help users in finding and booking their ideal vacation. Another goal is to turn this app into a profitable business venture. This may be attained by making use of the affiliate network. Furthermore, TravelMatch provides an interface for managers without a technical background to upload new images and destinations, et cetera.

2.4 Environment

TravelMatch is an application for smartphones and tablets. The supported operating systems include Android, version 4.1 "Jelly Bean" and up, as well as iOS, version 7.0 and up. Any other operating systems are not supported for this project. The app will support portrait and landscape orientations on tablets. On smartphones, only portrait orientation is supported and landscape orientation is blocked. As Android and iOS run on a wide range of devices, the TravelMatch app depends on the operating system to provide the proper interface and library functions.

The system will be used by the following users:

- **Users** People who find the app on either the Google Play Store or the iOS App Store.

  - **Guests** People who find the app and play around with it, without the direct intent to book a vacation.
  - **Indecisive travelers** Travelers who do not know where to go on a vacation and who want to receive an advice from the app for a destination.
  - **Budget travelers** Travelers who are on a budget and want to find a trip that is to their liking given their personal budget.

- **Managers** The people who manage the back end database, who will add images and tags that will be sent to the users.
- **Administrators** The people responsible for the back end server, who will solve any problems that arise during operation.
- **Developers** The people responsible for maintaining compatibility with future OS releases and who will improve the application in the future.

The characteristics of these users are specified in detail in Section 2.4 of the URD. [3]

2.5 Relation to other systems

TravelMatch is a standalone application and no other system depends on it. However, TravelMatch itself depends on other systems.

2.5.1 Affiliate Network

TravelMatch is connected to an affiliate network as a publisher to retrieve information about trips from advertisers. Advertisers can start an advertising campaign in one of several predefined categories. Publishers can join such advertising campaigns. This allow the publishers to receive product feeds from the advertisers. With these product feeds, publishers can attract potential customers and forward them to the advertiser. If the customer makes a purchase as a direct result of the publisher's advertisement, the publisher receives a share of the revenue. TradeTracker is an affiliate network that connects publishers with advertisers. TravelMatch acts as a publisher, because it requests product feeds from TradeTracker. When a user is interested in booking a particular trip, TravelMatch will use TradeTracker to obtain deeplinks to the travel agency's website. These deeplinks contain unique identifiers that allow TradeTracker to recognize TravelMatch as the publisher. Afterwards, TravelMatch can query TradeTracker with this unique identifier to check whether the user has booked the hotel.

2.5.2 Authentication providers

Users can register an account using one of several authentication providers. In the current scope, the first authentication providers is the TravelMatch back end server itself, which allows users to register via e-mail address and password. The other authentication provider is Facebook Login, which allows users to register via their Facebook account.

If the user chooses to make a new account, they have the option to log in with Facebook and authorize the TravelMatch app to use their account details. Upon successful authorization, a new user account will be created in the TravelMatch back end server. Afterwards, the user may log in to their TravelMatch account simply by logging into their Facebook account.

2.5.3 Analytics

The TravelMatch app will support an external analytics provider to monitor user behavior in the TravelMatch app. A manager can access the analytics data within the control panel of the external service itself. The TravelMatch app will use Google Analytics to monitor which screens are being viewed and for how long, among other items of interest. Google Analytics receives the analytics data

straight from the TravelMatch app on the user's device, as the app connects directly to the analytics service via the Internet.

2.6 General constraints

The TravelMatch app should be easy to use and understand by anyone, and it must be responsive. Furthermore, the app requires a constant connection to the back end server in order to work. The system should be designed to be reliable, maintainable and easily extendible. This last constraint is especially important since the customer intends to further develop the system after delivery.

2.6.1 Security

The TravelMatch app does not store sensitive information on the phone. However, since it uses authentication providers, the back end server will store sensitive information that should not be made public such as passwords. For this reason, the back end server will store all passwords in the database in a hashed and salted form to protect the sensitive information in the event of a database intrusion. The TravelMatch app transfers the unhashed password to the back end, but this transferral can be done over an encrypted channel. To connect with Facebook, the other authentication provider, more securely, we make use of the OAuth 2.0 protocol for communication. To ensure a safe connection with the back end, an SSL connection to the back end is provided.

2.6.2 Image sizes

Different sizes are saved per image, in order to optimize bandwidth usage when downloading images from the server and to display images on the client side with minimal artifacts. We can assume that these sizes will not change very often in the future. Therefore the image sizes are not linked to the image entities in the database model. Instead, each time an image file is changed, the versions for all sizes are generated in the database. These sizes are automatically created by expanding or contracting the original image until it completely fills the desired dimension. Then, each size is stored as a compressed .JPEG file using ImageMagick[28] and are saved with a certain pattern in their file name. From an image with original file name without extension N, width w and height h the cropped version is saved with the filename:

$$N + \text{"\_"} + w + \text{"x"} + h + \text{".jpg"}$$

This pattern is used to find deprecated sizes and to retrieve the images via the web server.

2.6.3 Extendibility

The system is meant to be further developed by iLysian, so the app and the back end must be easy to maintain and extend. Initially, the TravelMatch app will support external login via Facebook; in the future, it must be possible to add different authentication providers. Similarly, the back end must be able to use any affiliate network. Another requirement for TravelMatch is that the language is Dutch by default, but as the user base grows a release in foreign countries can be a profitable choice. For such an event, the language specific features should be extendible; there should be a language component that facilitates changing between languages. For the input of images the TravelMatch app displays, a CMS is created to add images with corresponding tags.

2.7 Model description

2.7.1 Domain model

The model of TravelMatch can be divided into two main components as represented in the domain model (Figure 2.1): the TravelMatch app and, within a marked frame, the back end server.

The TravelMatch app is the front end application that runs on Android and iOS smartphones and tablets. It will be used by the end users to search for a vacation. To obtain this cross-platform compatibility, we use the Ionic framework [22]. This framework allows developers to cross-compile for various mobile operating systems from a single codebase. Ionic itself makes use of the AngularJS framework, which is written in JavaScript.

The app requires a constant connection to the back end server in order to perform authentication, obtain destination recommendations and available hotels, and record analytics. The back end server, however, does not depend on the TravelMatch app. The back end server is used by managers, administrators and developers from iLysian to maintain the app. For the server software we use the Django framework, which uses the Python programming language. The server software runs on a Linux virtual server.

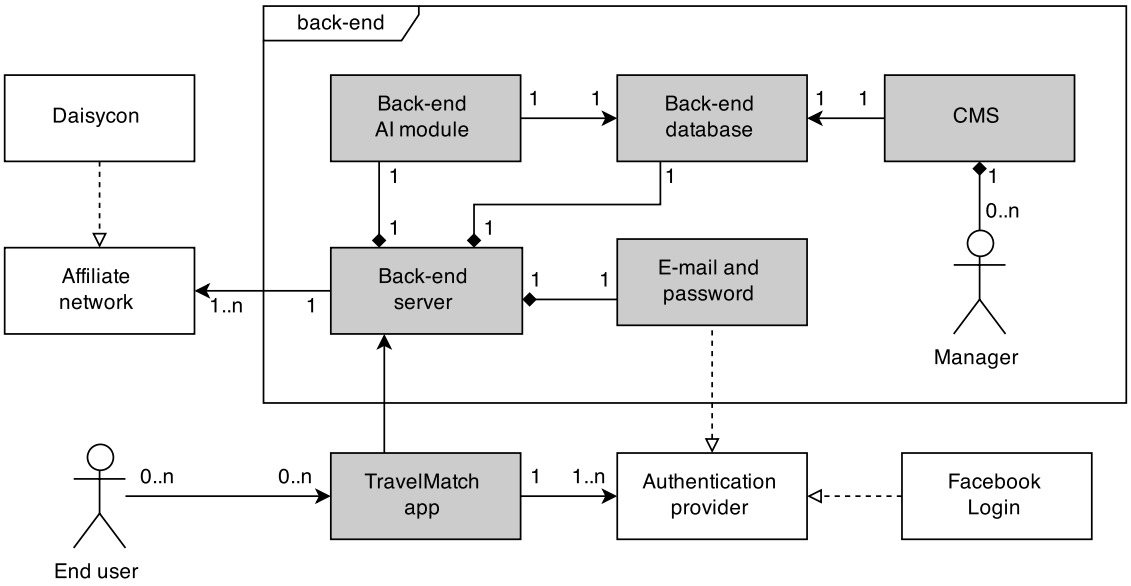

Figure 2.1: Domain model

2.7.2 Model-View-Controller model

The TravelMatch app is written in the Ionic framework, which in turn uses AngularJS, a JavaScript framework. AngularJS follows a Model-View-Controller model, as depicted in Figure 2.2. On the front end, this Model-View-Controller model is implemented as follows:

- **View**: The view portion of the app is implemented as HTML and CSS files. It is split up into the main view – a base HTML file that includes all other files – as well as template view files for every screen in the app.
- **Controller**: The controller is implemented as AngularJS-flavored JavaScript files. These controller files call services.
- **Model**: In practice, the model in the TravelMatch app is largely dependent on the controller in order to function.
- **Services**: These services are responsible for communication with the server via API calls.

On the back end, the Model-View-Controller is implemented as follows:

- **Controller**: The web server controller is responsible for handling incoming API calls. In doing so, it communicate with the database model and the CMS view.
- **Model**: All data is stored inside the database.
- **View**: The CMS gives a view of the data, and can retrieve the data by calling the controller.

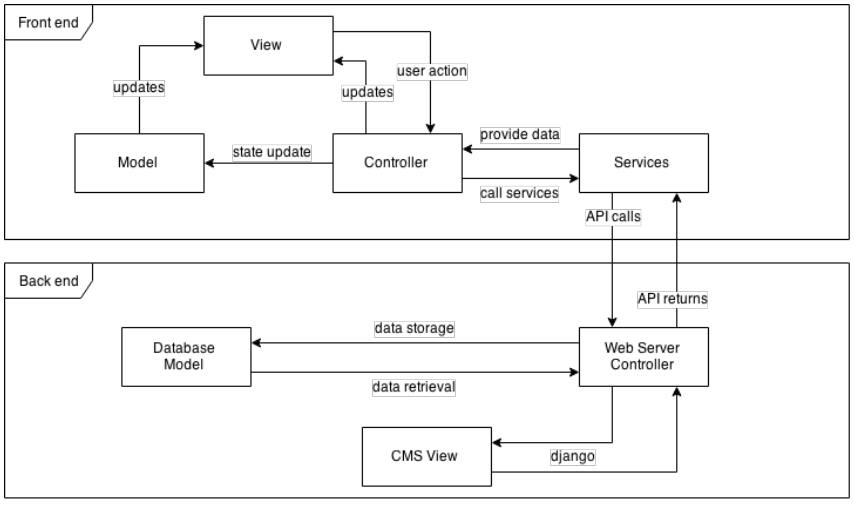

Figure 2.2: Model-View-Controller model

2.7.3 Data model

An important subcomponent of the back end server is the database. We use Django's built-in database manager, which is an object-oriented database system. The database storage system that is used by Django can easily be adapted in the future by changing the settings of Django. The administration module is an easy-to-use CMS functionality of Django that may be used by iLysian employees to manage all the data in the database. [25]
The ER diagram of the database has a structure as shown in Figure 2.4 with a legend as depicted in Figure 2.3. Brief explanations for all non-trivial entities and attributes of our model follow.

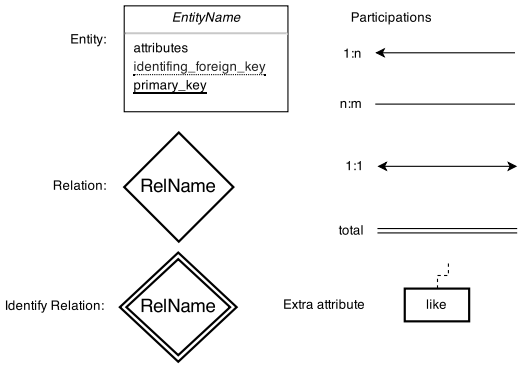

Figure 2.3: Database ER diagram legend

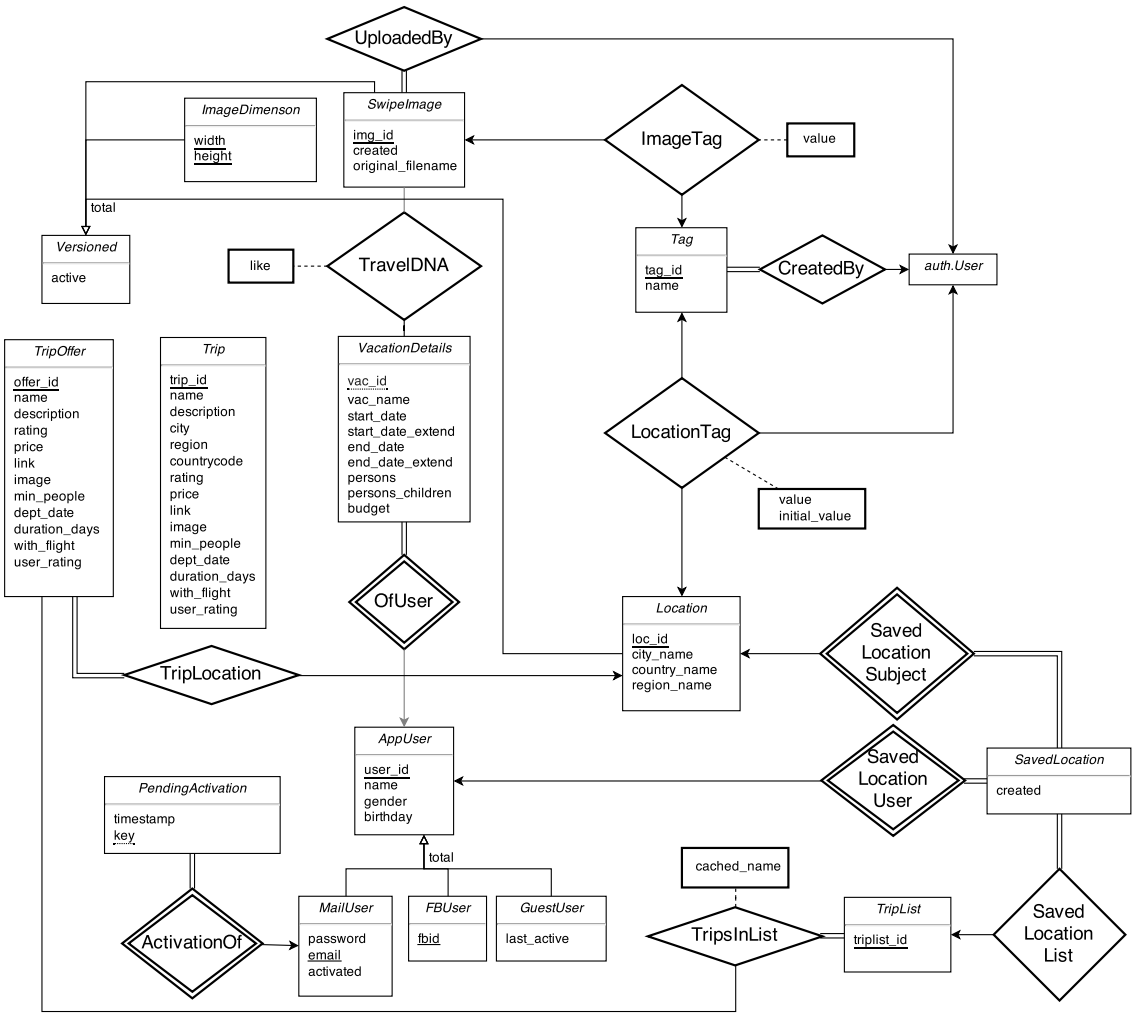

Figure 2.4: Database ER diagram

- **AppUser**
  This entity can be extended via different authentication mechanisms to represent an end user of the app.
  - *user_id:int* A number that uniquely identifies that particular user.
  - *name:string* The first and last name of the user.
  - *gender:int* The gender, where each value corresponds to the following meaning:
    - **0**: Not set yet;
    - **1**: Male;
    - **2**: Female;
    - **3**: Explicitly set to none.
  - *birthday:date* The birth date of the user.
- **MailUser**
  A user that is authenticated via email and password.
  - *password:string* The hashed and salted password.
  - *email:string* The user's e-mail address.

  - *active:boolean* Indicates if the account is activated via mail.

- **PendingActivation**
  Whenever a mail user is created and an activation email is sent, the key that must be used to activate the user is stored here.
  - *timestamp:datetime* The timestamp of when the confirmation email was sent.
  - *key:string* The generated identifier for activation. These keys are unique.

- **FBUser**
  When an user logs in via Facebook, this entity couples their Facebook ID to our user profile.
  - *fbid:string* The unique identifier for a Facebook authenticated user, as specified by Facebook. [27]

- **GuestUser** The user can also choose to not log in. In this case, the client creates a guest account.
  - *last_active:datetime* This keeps track of the last time the authentication of the guest user was used. This can be used to remove guest account of people that have removed the app.

- **VactionDetails**
  All interactions with users go per vacation details instances so that the different profiles of each separate holiday do not interfere with each other.
  - *vac_id:integer* The identifier of a user's vacation details.
  - *vac_name:string* The given name of the vacation details, if set by the user.
  - *start_ date:date* The starting date of the vacation.
  - *start_ date_extend:integer* The margin of the starting date, in days.
  - *end_date:date* The end date of the vacation.
  - *end_date_extend:integer* The margin of the end date, in days.
  - *persons:integer* The amount of adult persons for the vacation.
  - *persons_children:integer* The amount of children for the vacation.
  - *budget:integer* The amount of money appointed to the vacation per person, or zero when the "Surprise me!" function is selected.

- **TravelDNA**
  The Travel DNA of a user is stored in the database as separate entries per image and vacation details. The Travel DNA is modelled as a relation between a user's vacation details, an image and a liking or disliking of that image. This way, only the most recent decisions per image are recorded.
  - *like:boolean* A statement of liking or disliking an image.

- **SwipeImage**
  An image that is uploaded via the CMS, to be used for the interest analysis. The SwipeImage is also connected to some author to record which of the Django users uploaded the image.
  - *img_id:integer* The unique identifier for the image.
  - *created:datetime* The date and time the image was added to the database.
  - *orginal_filename:string* The relative path from the MEDIA_ROOT to the image on the server. [1] Django normally uses the same filename as the uploaded file and automatically appends some hash when this gives a conflict.

[1] The MEDIA_ROOT is a variable of Django that points to some folder where all the images are stored. These can then be retrieved via the web interface by using the MEDIA_URL variable of Django.

- **ImageDimension**
  This is an entity that stores all different image sizes that are on the server. Section 2.6.2 explains how these are related to the images.
  - *width:integer* The width in pixels.
  - *height:integer* The height in pixels.
- **ImageTag**
  This stores all tags' values per image.
  - *value:integer* The value of the tag for this image that is entered by a Django user.
- **Tag**
  These are all the available types of tags.
  - *tag_id:integer* The unique identifier for the tag.
  - *name:string* The name of the tag.
  - In addition, the creator of the tag is also recorded.
- **LocationTag**
  This stores all the values of tags per location.
  - *value:integer* The value of some tag for some location that is used and changed by the AI
  - *initial_value:integer* The initial value of some tag for some location that was entered by a Django user.
- **Location**
  This entity represents a holiday destination that can be given as an advice.
  - *loc_id:integer* The unique identifier of a location.
  - *city_name:string* The city name of the location.
  - *country_name:string* The name of the country where the city is situated.
  - *region_ name:string* The name of the region where the city is situated.
- **Trip**
  This is a trip offer from the affiliate network that is downloaded, but not used. These trips are downloaded from the affiliate network using the corresponding parsers.
  - *id:integer* Some Django unique identifier that is only used as primary key.
  - All the values retrieved from the affiliate network.
- **TripOffer**
  This is a trip offering from the affiliate network that is used by the AI. In contrast to the Trip entity, this entity is linked to some location in the database. The Django users can fill this entity from the **Trip** entity if and only if the location is already present in the database.
  - *loc_id:integer* The unique identifier, which does not have to be equal to the ID in the corresponding Trip object.
  - All the values retrieved from the affiliate network that the app wants to display.
- **SavedLocation and its relations**
  A saved location is a weak entity with its primary keys in Location and AppUser. It also contains the list of trips that was displayed to the user. This list can contain a number of **TripOffer** entities. Because these entities can be updated via the API, a cached name is also stored for each **TripOffer**.

  - *created:datetime* The timestamp of creation.
  - *cached_name:string* The last displayed name of the hotel.

- **auth.User** [2]
  This represents the Django default user system. It manages the user accounts of the CMS. It can also be used to set permissions for altering certain entities.

- **Versioned**
  All entities that implement this superclass have an extra versioning boolean. The default value, when instances are created, is $False$. The AI and API will then ignore these instances. As soon as the variable is set to $True$, it can be used by the other components. Only the CMS uses the entities when they are not active.

2.7.4 Class Diagrams

Class diagram of the affiliate network

A detailed description of how the affiliate network is parsed and stored in the database can be found in Figure 2.5. Another controller can interact with the Affiliate object by adding parsers to it. These concrete parsers must fill in a factory method, which returns their respective ConcreteAttrib. Furthermore, specific feed URLs can be added to specific parsers with their respective feed-specific properties.

- **Affiliate** The object that manages all parsers and which other controllers can interact with. It contains the following list of Parsers.

  - *_all_parsers* A dictionary of all parser names mapped to their respective parser objects.

  The affiliate object can be called with the following functions:

  - *add_parser* Adds a parser to the Affiliate object.
  - *show_parsers* Shows all parsers currently inside the Affiliate object.
  - *remove_parser* Removes a parser from the Affiliate object.
  - *add_feed* Adds a feed to a parser inside the Affiliate object.
  - *show_feeds* Shows all feeds stored inside a parser.
  - *remove_feed* Removes a feed of a parser.
  - *store_all* Retrieves information from all feeds inside all parsers and stores the required information inside the database.

- **AffiliateNetwork** The affiliate network to retrieve all trip information from.

- **Feed_url** The URL of a feed of an affiliate network.

- **FeedProperties** Feed-specific properties, such as whether all trips are with or without a flight. It currently only has one property, but later more can be added:

  - *with_flight* Whether all trips of the feed have the flight included.

- **AffiliateParser** This Parser class contains all functions regarding handling and parsing all feeds that resides in this parser. It contains the following dictionary:

  - *_feed_urls* A list of the feeds inside this parser object mapped to their respective properties.

  Furthermore, the Parser contains some functions for feed administration and parsing them:

  - *add_single_feed* Adds a single feed to the Parser object.

[2] This is in the same syntax as Django to avoid confusion with our user model.

  - *show_feeds* Shows all feeds that resides inside this Parser object.
  - *remove_feed* Removes a feed from that is inside this Parser objects.
  - *process_all* Processes all feeds inside the Parser object by retrieving the ones inside this feed and storing the results in the database.
  - *_process_single* A private function that retrieves the information of a single feed and stores the resulting affiliate info inside the database.
  - *_get_root* Returns the root of a parsed XML tree from a feed url.
  - *_find_and_add_all_xml_attributes* Finds all XML attributes and adds their respective values inside the Attrib object.
  - *_find_and_add_xml_elements* Finds all XML elements and adds their respective values inside the Attrib object.
  - *_attribute_adt* An abstract factory method, whose implementations should return the concrete attributes ADT used by the instance of the concrete parser.

- **ConcreteParser** A concrete parser object for an affiliate network. It should consist of the following a unique name:

  - *parser_name* A unique name specific to this concrete parser.

  Furthermore, it should implement a factory method:

  - *_attribute_adt* Implements the factory method to return the concrete parser method used by this concrete parser object.

- **TripAttrib** The TripAttrib object consists of all XML variable names inside the feed and all (must have) model field names. Furthermore, it keeps track of all values that need to be stored in the database and stores the entry inside the database. It contains the following lists and dictionaries:

  - *_model_variables* All model field names that were specified inside the Trip model.
  - *_must_have_model* All must have model field names as specified by the client.
  - *_model_to_attributes* Internal dictionary that maps all model field names to their respective attributes. This dictionary is eventually stored inside the database.
  - *_xml_to_model* Internal dictionary that maps all xml variables to their respective specified model variables.

  This TripAttrib objects provides the following functions for manipulation of its variables:

  - *in_xml_keys* Returns whether an object is inside the defined XML variable.
  - *get_xml_keys* Returns a set of all XML variables specified in the concrete attributes ADT object.
  - *add_attribute_using_xml* Adds an attribute or value that belongs to a particular XML variable.
  - *add_attribute_using_model* Adds an attribute or value that belongs to a particular model variable.
  - *store_entry* Stores the entry that currently resides inside the _model_to_attributes dictionary into the database.
  - *_check_for_discard* Returns whether the current entry should be discarded. This is the case if any must-have model variables are set to $None$.
  - *_add_entry* This function is used by a subclass to add entries to this object with the specific XML variable names used inside the feed with their respective model names and default values if applicable.

  - *_get_correct_attribute_value* Gets the attribute or value that corresponds to the given model variable.
  - *_get_correct_date_format* Function to convert the date format DD-MM-YYYY to the required Django date format YYYY-MM-DD if applicable.
  - *_is_rep_ok* Internal check to look if the representation variant is still OK.

- **ConcreteAttrib** An attributes ADT specific to the affiliate network where the required XML names have been initialized with their respective model names and default values using _add_entry.
- **Model** The database where all information is stored.
- **Controller** Another controller that can create and call an Affiliate object.

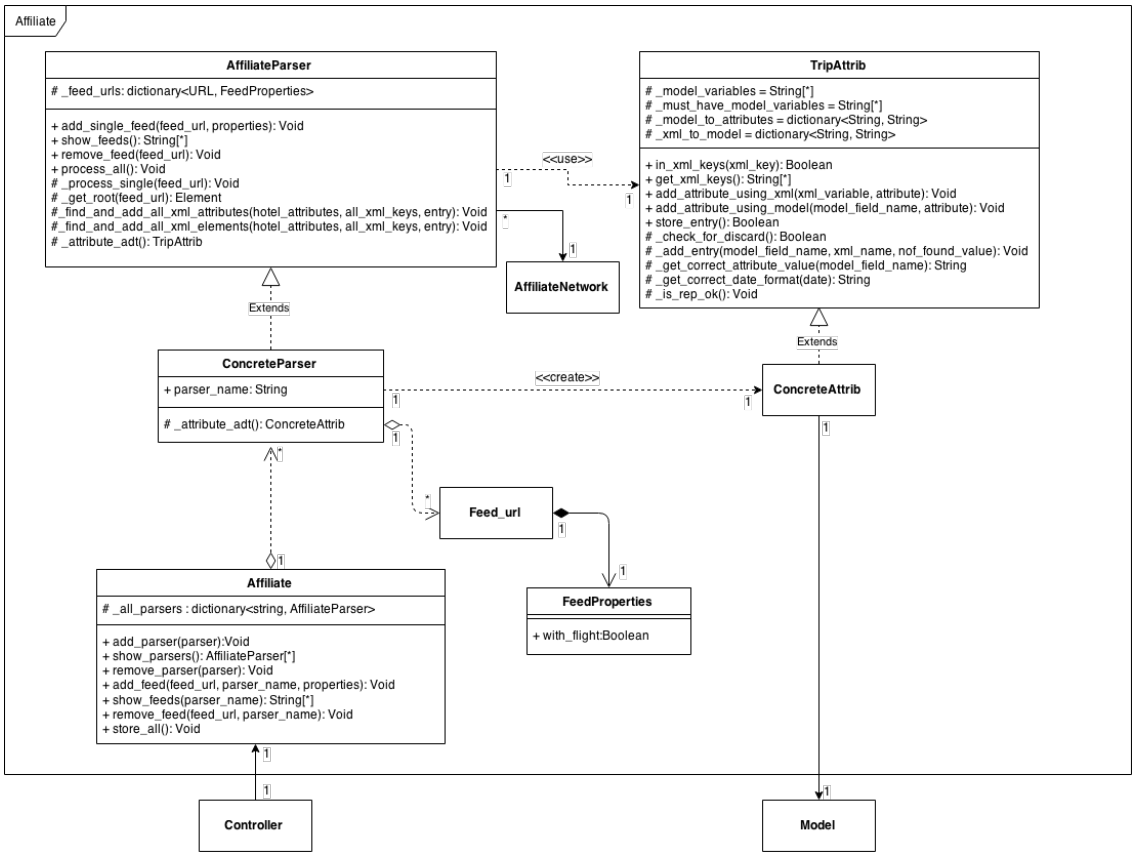

Figure 2.5: Class diagram of the affiliate network

Class diagram of the TravelMatch app

A diagram showing the setup of the front-end with the functionality.

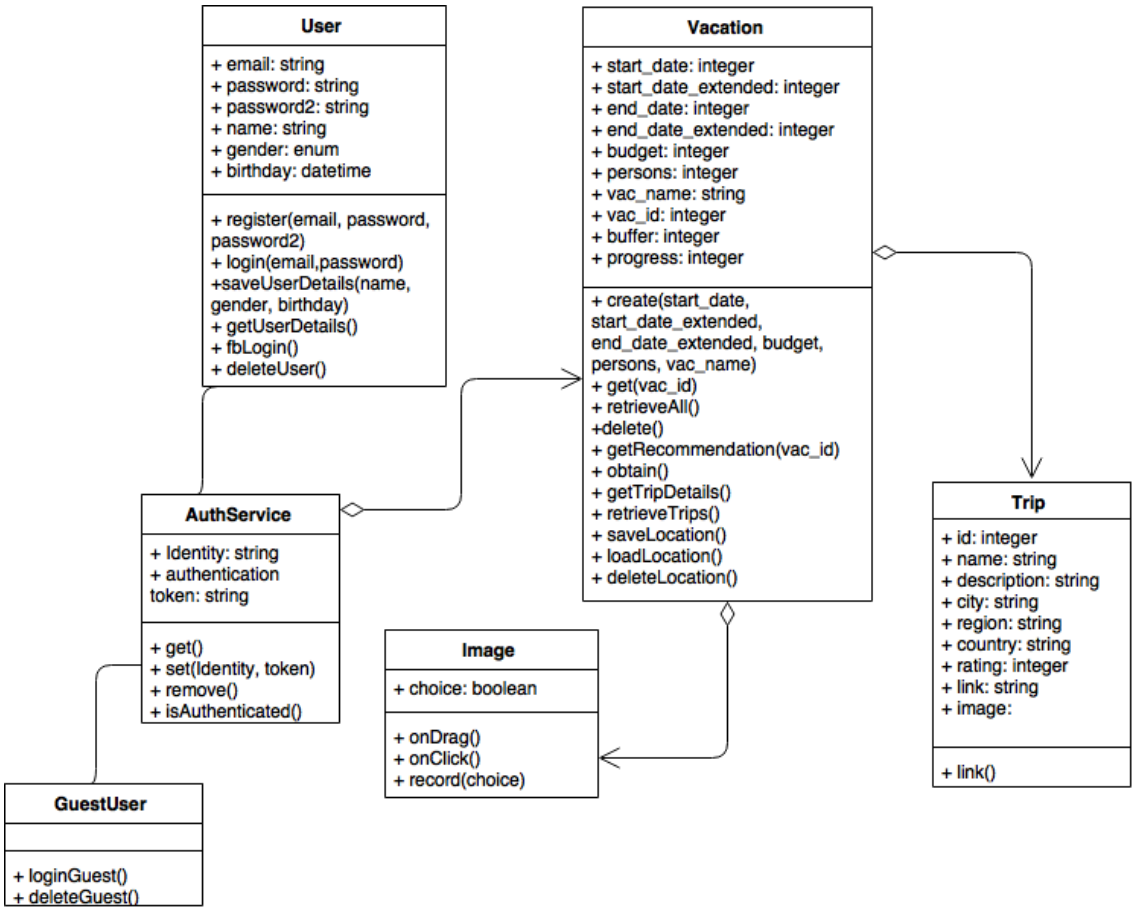

Figure 2.6: Class diagram of the TravelMatch app

- **AuthService** The TravelMatch app needs to store and retrieve an authentication token for the communication with the API.
  - *Identity* An attribute to hold the identity of the user.
  - *token* A unique identifier used to communicate with the server.

  This class provides the following functionality:
  - *get* The functionality to obtain the token and Identity.
  - *set* The functionality to change the token and Identity.
  - *remove* The functionality to remove the token and Identity.
  - *isAuthenticated* The functionality to check if a user is still authenticated.
- **GuestUser** The TravelMatch app supports a guest user. Guest users have less functionality than normal users.
  This class provides the following functionality:
  - *loginGuest* The functionality to obtain the token from the back end.
  - *deleteGuest* The functionality to remove the guest user.
- **User** The TravelMatch app supports users. Users have more attributes and functionality than guest users.

  - *email* An attribute to hold the entered email address of the user.
  - *password* An attribute to hold the entered password, which is hidden by default.
  - *password2* An attribute to hold the repeat entered password.
  - *name* An attribute to hold the name of the user.
  - *gender* An attribute to hold the gender of the user.
  - *birthday* An attribute to hold the birthday of the user.

  This class provides the following functionality:

  - *register* A function to register; it uses the attributes of email, password and password2. It will send an API request to store it in the database.
  - *login* A function to login, it uses the email and password attributes. It will send an API request to obtain the authentication token.
  - *saveUserDetails* A function to modify the user details. It uses the attributes of name, gender, birthday, as well as the authentication token from the AuthService. It then uses the token to make the API request.
  - *getUserDetails* A function to retrieve the user details. It uses the stored authentication token of the AuthService to make the API call.
  - *fbLogin* A function to call on a browser or app to authenticate with Facebook.
  - *deleteUser* A function to remove the user from the device and back end database.

- **Vacation** The class for handling the vacations. It will handle the interest analysis and the recommendations.

  - *start_date* An attribute to hold the starting date of the vacation.
  - *start_date_extended* An attribute to hold the extend of difference of start_date.
  - *end_date* An attribute to hold the return date of the vacation.
  - *end_date_extended* An attribute to hold the extend of difference of the entered end_date.
  - *budget* An attribute to hold the amount of money is allocated per person.
  - *persons* An attribute to hold the total amount of persons.
  - *vac_name* An attribute to hold the name of the vacation being made.
  - *vac_id* A unique identifier to distinguish between different vacation details.
  - *buffer* An attribute that holds the amount of image stored on the TravelMatch not judged yet.
  - *progress* An attribute that holds the total amount of images needed for, and how far the User is with their interest analysis.

  This class provides the following functionality:

  - *create* The functionality to create vacation details for a user.
  - *get* The functionality to retrieve the vacation details by ID.
  - *retrieveAll* The functionality to retrieve all the vacation details in the database for a specific user.
  - *delete* The functionality to delete vacation details.
  - *getRecommendation* The functionality to generate a recommendation and retrieve trips from the API.
  - *obtain* The functionality to retrieve images from the API.
  - *getTripDetail* The functionality to retrieve more details about a trip.

  - *retrieveTrips* The functionality to retrieve trips from the API.
  - *saveLocation* The functionality to save the viewed location.
  - *loadLocation* The functionality to load the saved location.
  - *deleteLocation* The functionality to delete a previously saved location.

- **Image** The system will show a lot of images that will be judged. The images need to be able to hold whether they were (dis)liked.

  - *choice* An attribute to hold the choice made about an image.

  This class provides the following functionality:

  - *onDrag* The functionality to drag the image to (dis)like it.
  - *onClick* The functionality to be able to click on a (dis)like to record the choice.
  - *record* The functionality to communicate with the API to record the choice.

- **Trip**

  - *name* An attribute to hold the name of the trip.
  - *description* An attribute to hold the description of the trip.
  - *city* An attribute to hold the trip city.
  - *region* An attribute to hold the trip region.
  - *country* An attribute to hold the trip country.
  - *rating* An attribute to hold the rating of the trip.
  - *link* An attribute to hold the deeplink needed to book the trip.
  - *image* An attribute to hold an image of the trip to show its facilities.

  This class provides the following functionality:

  - *link* The functionality to redirect the user to the booking site.

2.7.5 Message sequence diagrams

Register via e-mail

A user fills in their e-mail address and password in the GUI. When they submit their information, this data is sent to the back end server. The back end server, in turn, sends the data to the database to check if the e-mail address is valid and if the email already exists in the database. If the e-mail address already exists in the database, the database sends its found response back to the server, with which the server sends an error to the GUI regarding an already existing e-mail address. In the other case, the database responds with a "not found" response. In this case, the server sends a request for an inactive account to be created, and a request to create a pending activation to the database. This is followed by sending an activation e-mail. If the e-mail is sent, this result is sent back to the GUI, in which the GUI responds to the user with the feedback; either an error or success.

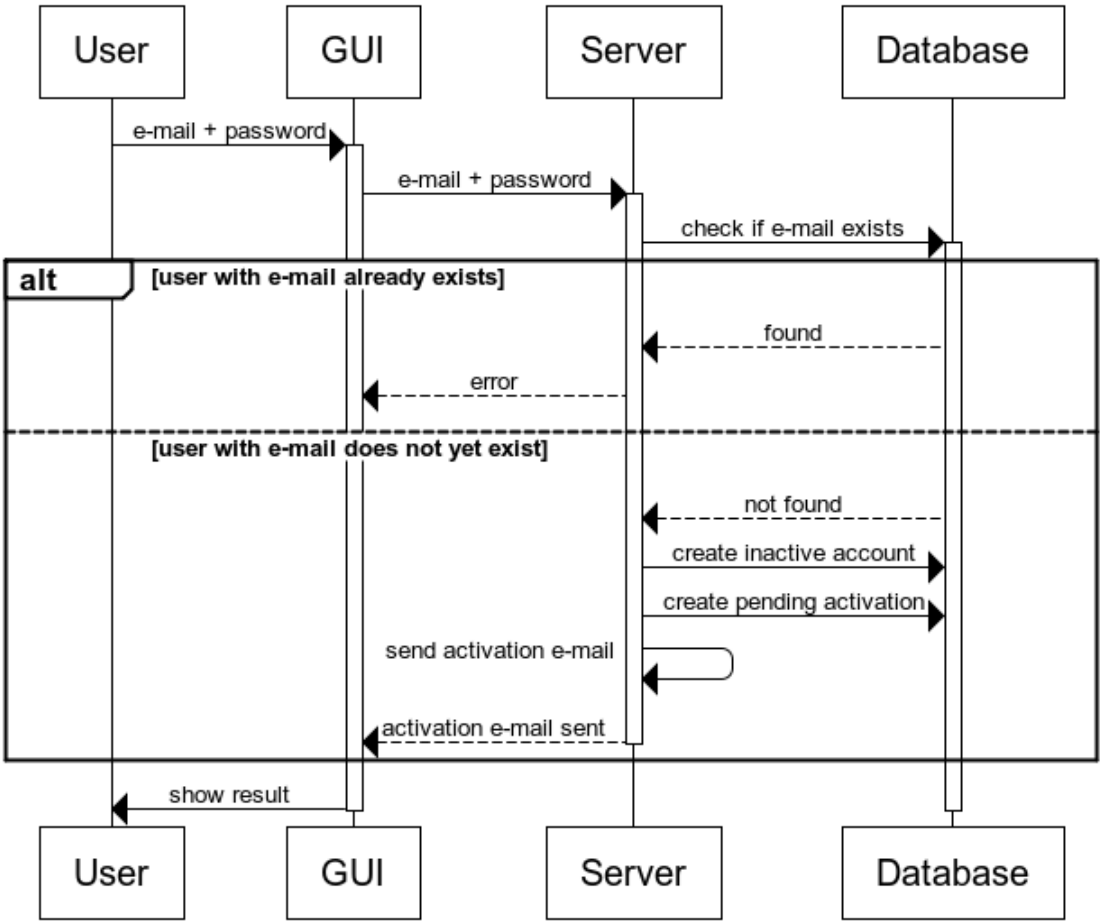

Figure 2.7: Registering an account via e-mail

Activate account via e-mail

If a user opens the link to activate their account, their browser sends the activation key to the server. The server sends a search request to the database for the activation. In this case, the database sends a user ID to the server when a pending activation exists, and the server returns a request to activate the account. After that, the server sends a response to the browser that the account is activated. If the pending activation does not exist, a not found response is given from the database to the server. The server then relays this an error to the browser. Finally, the browser shows the results to the user.

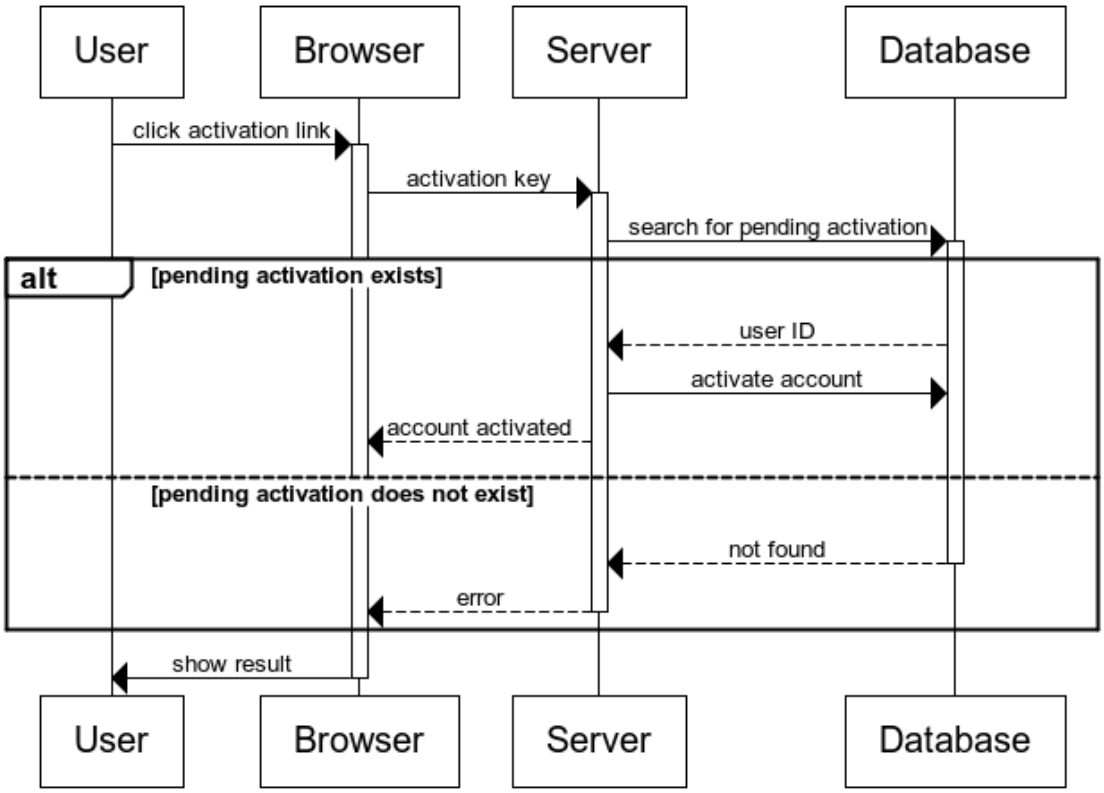

Figure 2.8: Activating an account via e-mail

Log in via e-mail

Similar to the registration, a user enters their e-mail address and password in the GUI and when a user submits their entry, this information is sent to the server, which is relayed to the database. The database checks for the existence and activation of the e-mail address and afterwards a check for the password, if the e-mail address is activated and found. If all of those were correct, the server gets the user ID as response and responds with a create session request to the database. The database then returns an authentication token which is traversed to the GUI. If one of the validations of e-mail address and password is invalid, a "not found" response is sent to the server, which in turn sends an error to the GUI. From the response of the server, the GUI will show the results.

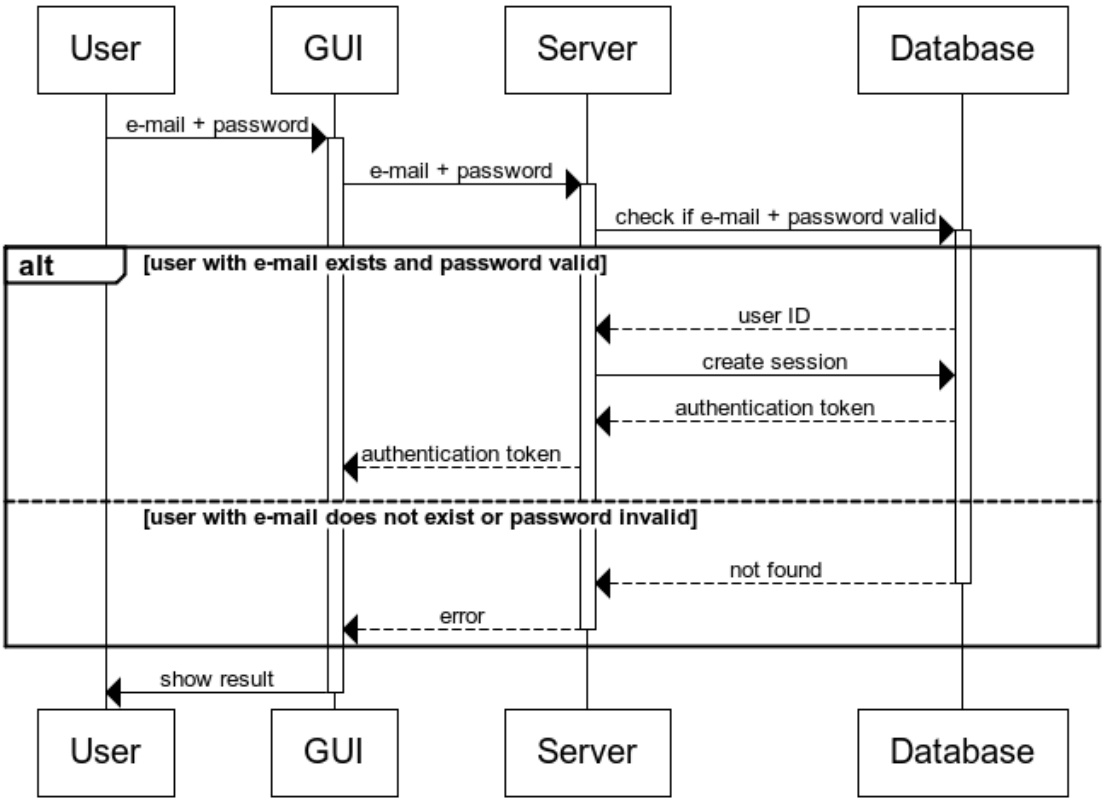

Figure 2.9: Logging into an account via e-mail

Register or log in via Facebook

The registration or log in via Facebook starts with a call of the user to the GUI to use the Facebook login. This request is sent to Facebook to resolve logging in. Facebook returns a Facebook ID and a token to the GUI which in turn sends it to the server, which in return sends it back to Facebook to check validity. If Facebook sends a success to the server, the server sends a request to the database to check if the account exists. If the account does not exist a new request is sent to create an active account. In either case, a success is returned to the GUI. If the token is found to be invalid, Facebook returns failure to the server which will then relay a failure to the GUI. In the last case Facebook, sends a failure response to the GUI for the authentication for Facebook. Finally, the result is communicated from the GUI to the user.

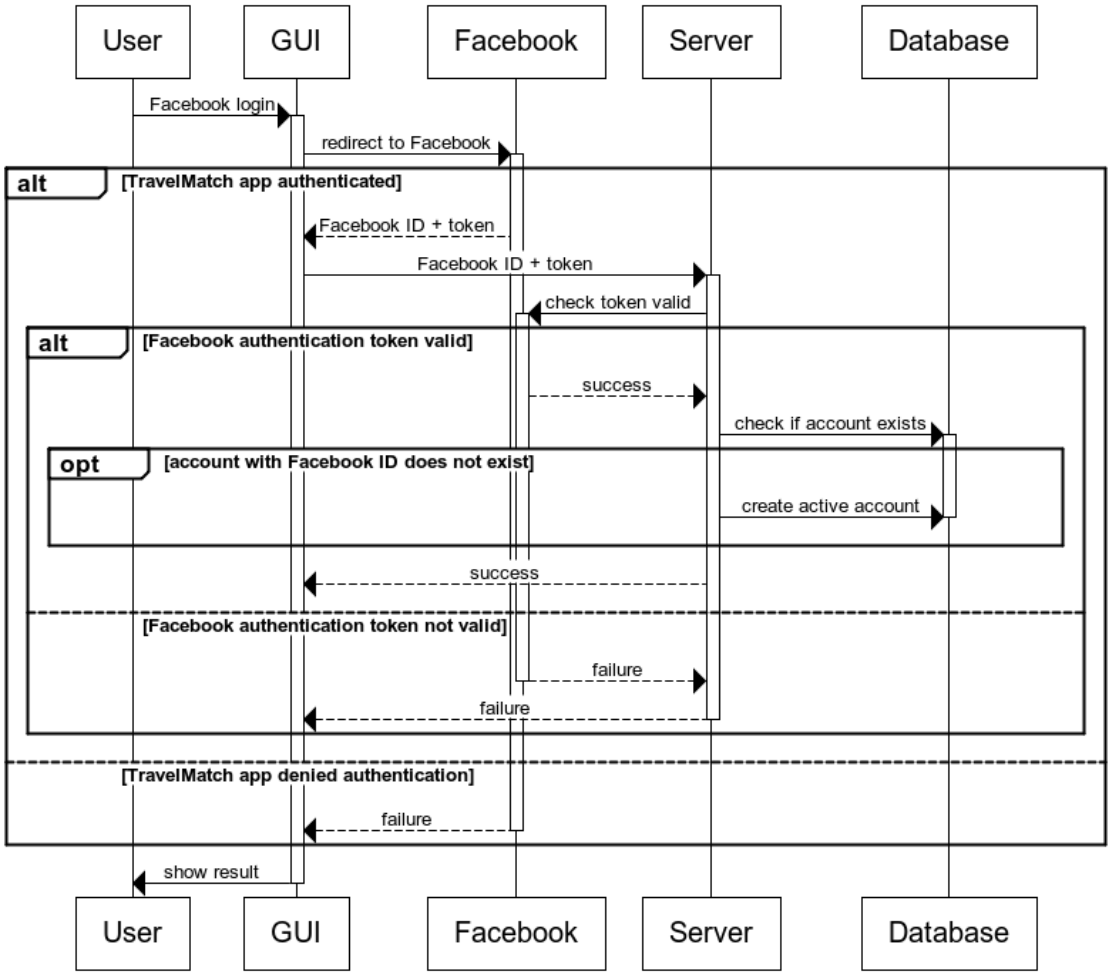

Figure 2.10: Registering or logging into an account via Facebook

Delete user account

To delete a user account, the user requests the GUI to delete their account. This request is send to the server with their authentication token and identity. The server validates this data with the database. If the database can find the account, it is removed and a success message is send to the Server. The server sends this to the GUI. If the database cannot find the account, it responds a "not found" message to the server, which is forwarded to the GUI. Finally, the user gets feedback on the request to delete the account.

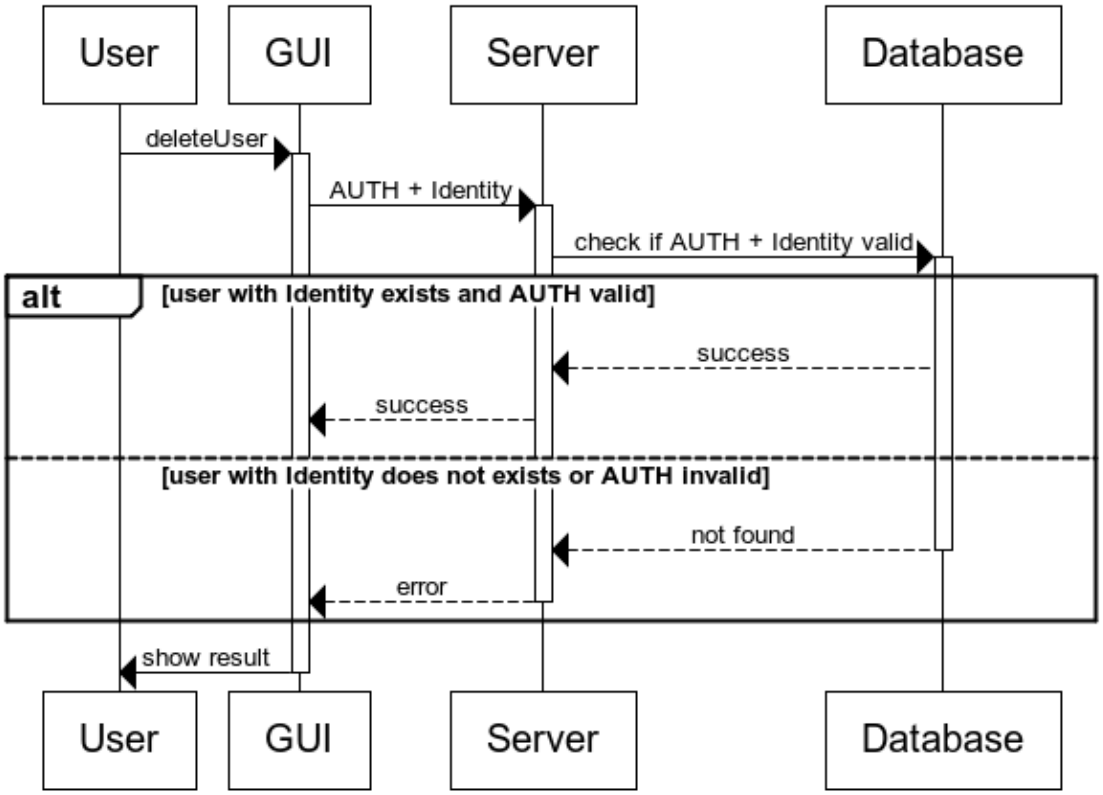

Figure 2.11: Delete user

Get user details

The retrieval of user details starts with the user opening the tab of the user details. The precondition for this action is that the user is logged in. Because of this, the GUI can send the authentication token to the server. This will trigger the server to send the token to the database which checks the validity. If the token is valid, the name, gender and birthday are returned to the server, which in turn responds to the GUI with the same. If the token is invalid, a "not found" response is returned to the server, which is relayed as an error to the GUI. Finally, the GUI shows the found results of name, gender and birthday; empty if not filled in yet.

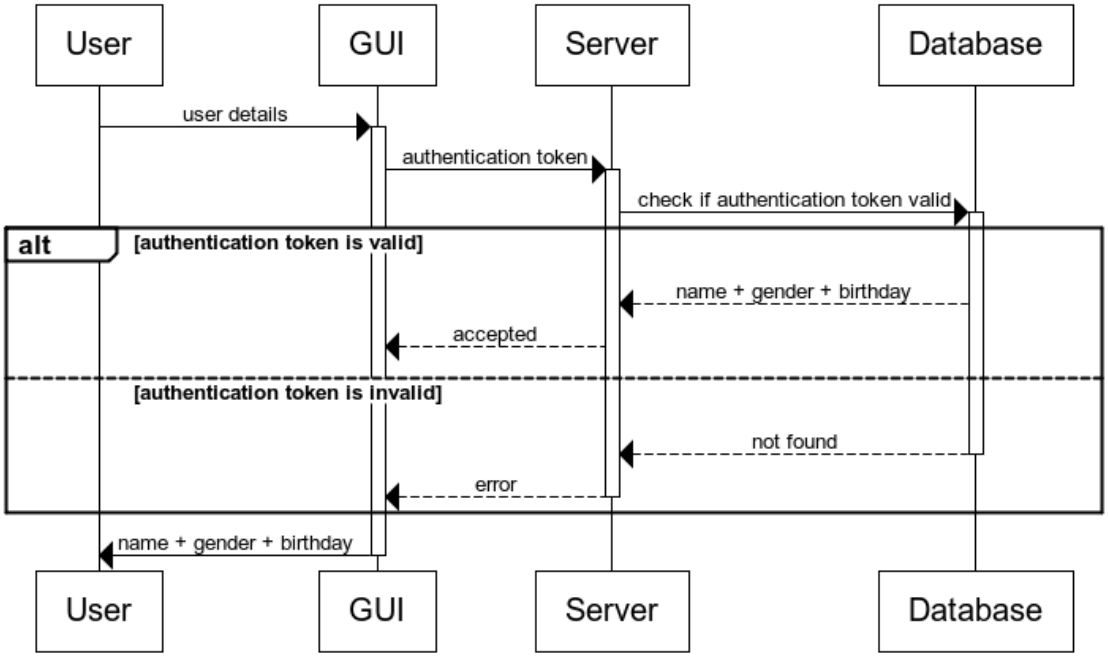

Figure 2.12: Get user details

Update user details

Updating the user details is initiated by the user when the submit button is pressed. A user has entered any of the available data about gender, birthday and name. The GUI takes the obtained data, adds the authentication token and sends it to the server. The server in turn checks the authentication token with the database. Next, if the authentication token is valid, the database responds with "accepted". In turn, the data of name, gender and birthday is send to the database to be stored, to be accepted by the database. Upon which the database sends the acceptance to the server, which relays to the GUI. In the case that the authentication is invalid, a "not found" response is given to the server, upon which the server returns an error to the GUI. Finally the GUI displays the result of the request to the server.

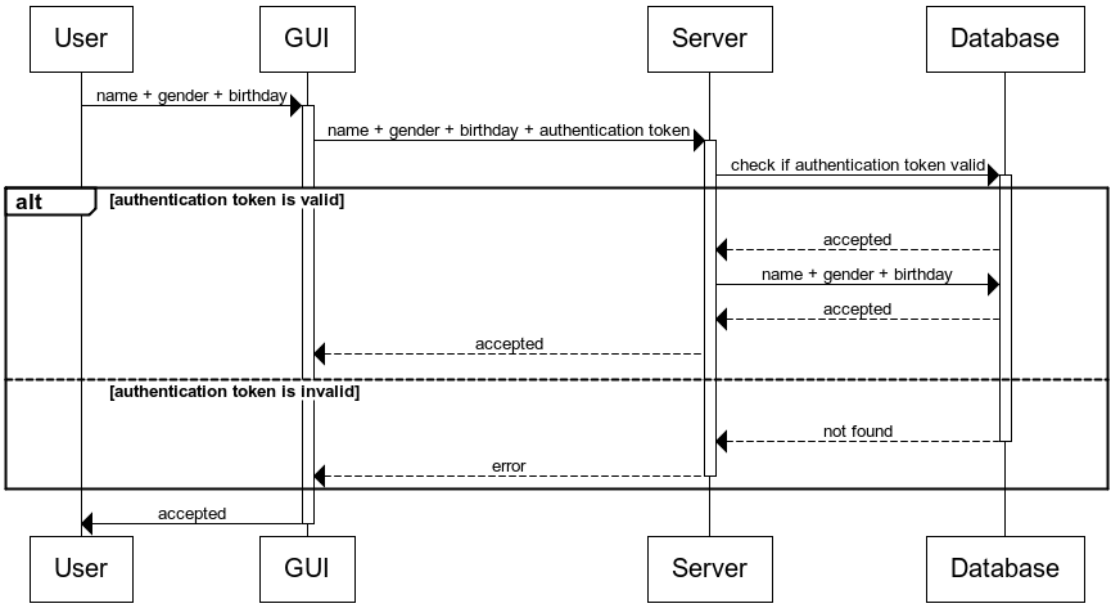

Figure 2.13: Update user details

Login guest user

If a user wants to use the TravelMatch app without logging in, the user requests the GUI to authenticate as a guest user. The GUI sends the request to the server with the device ID; this is forwarded to the database to validate the device ID. If the device ID is valid, the database responds with an authentication token to the server, which the server forwards to the GUI. If the device ID is invalid, the database responds with an error message to the server, which relays this to the GUI.

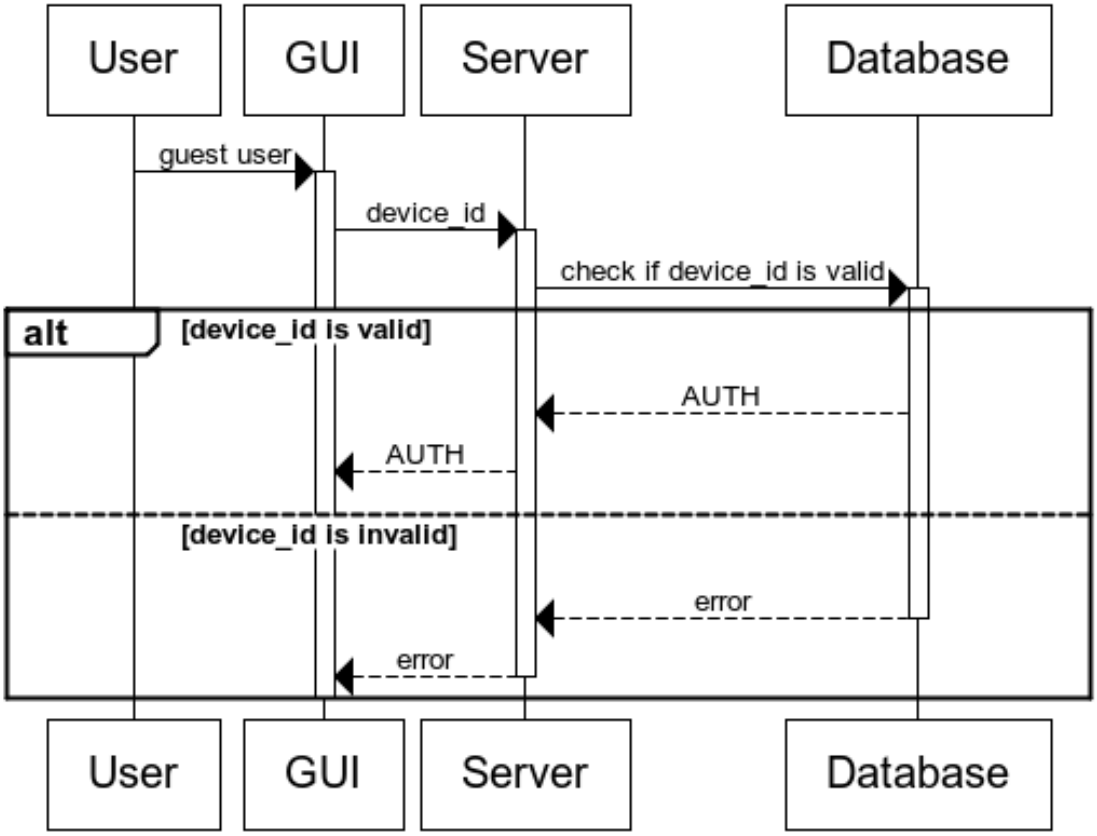

Figure 2.14: Guest user login.

Delete guest user

If a user logs out as a guest user, this request is sent to the GUI. The GUI attaches the device ID to the request to send it to the server. The server requests the database to validate the device ID. If the device ID is valid, the database responds with a success message to the server, which forwards this to the GUI. If the device ID is invalid the database responds with an error message to the server, which forwards this to the GUI. Finally, the GUI will show the result.

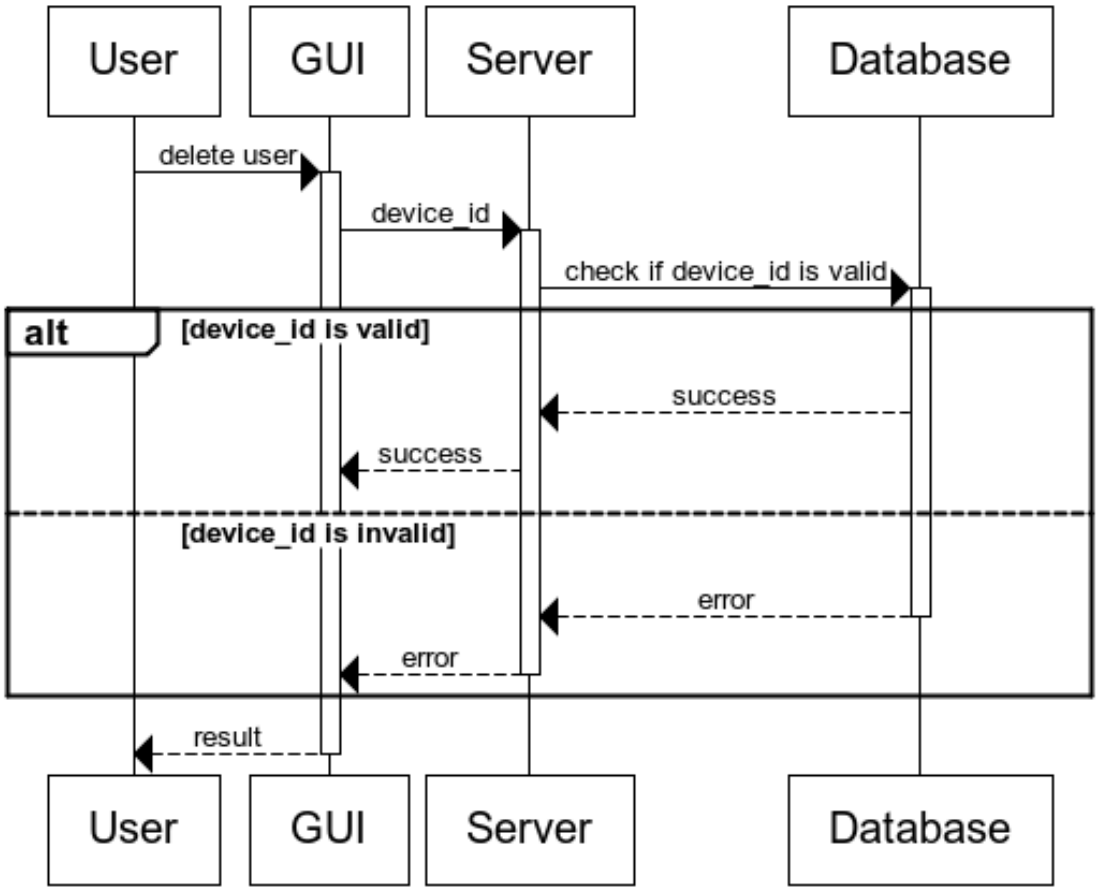

Figure 2.15: Delete guest user.

Affiliate setup

To retrieve data from the affiliate service, first an affiliate object has to be created. The Affiliate object is used for all matter related to the affiliate networks. Afterwards, Parsers can be created and added to this affiliate object. Now, feeds can be assigned to specific parsers by calling them via their parser_name. While adding these feeds, specific feed-related attributes can be added as properties. These are called FeedProperties, where the properties are assigned during the creation of this object. First of all, an affiliate object is created. Secondly, a Parser object is created and the Parser is added to this affiliate object. Thirdly, an feed is added to this Parser object by creating its FeedProperties and assigning it to this parser by its parser_name. Furthermore, the feeds of this Parser object are shown and finally this feed is removed from the Parser.

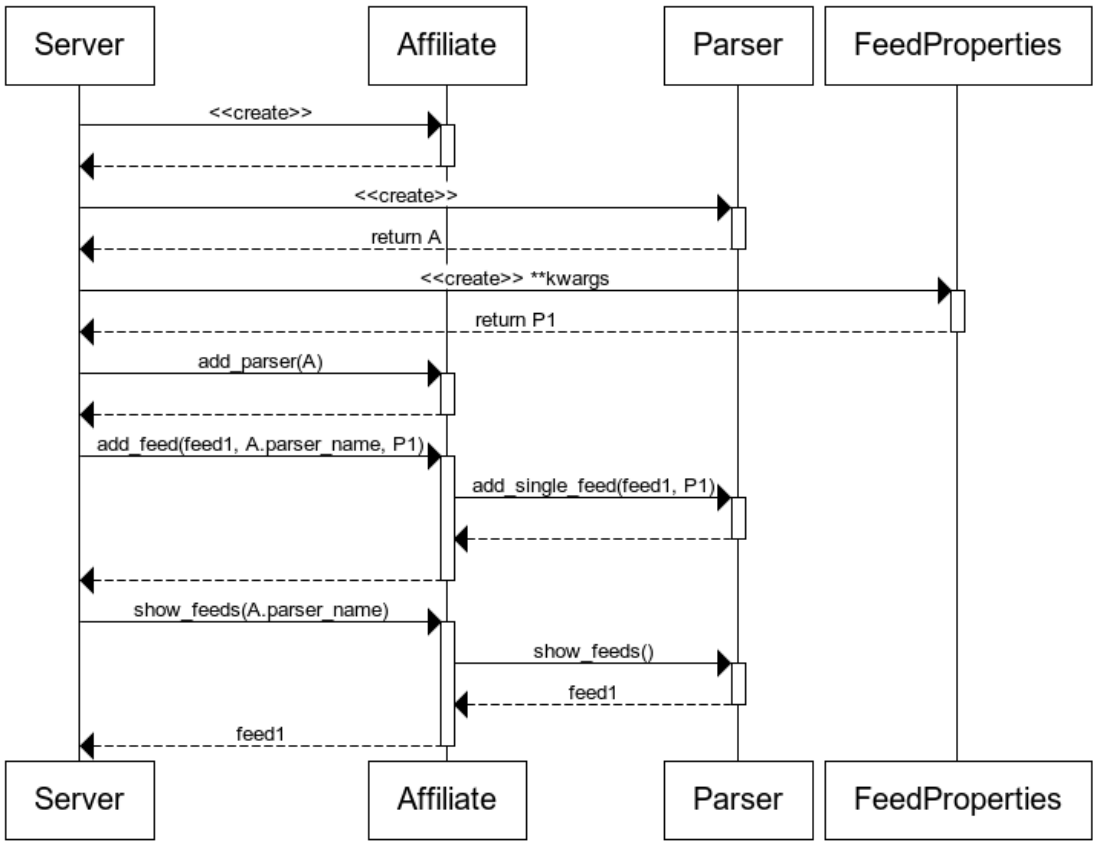

Figure 2.16: Affliate setup elaborated

Affiliate retrieve and save feed

The function store_all() is used to retrieve the data of all feeds of all parsers and store the information into the database. Inside this function, there is a call to process_all() in each Parser object, the results of which are stored inside the affiliate object. This is the first loop shown in figure 2.17 In these Parser objects, all feeds will be processed using the function process_single(). This is the second loop shown in figure 2.17. The processing of a single feed consists of:

1. Retrieving the raw xml data from the affiliate server.
2. Parsing this data.
3. For each entry inside this feed:
   (a) Adding all needed xml_attributes which were specified in the TripAttrib object.
   (b) Adding all needed xml_elements which were specified in the TripAttrib object.
   (c) If all vital entries were defined then adding the feed information to the model and saving, otherwise dropping the feed.

Note that there are two different steps in finding the correct XML names and storing their respective values inside the TripAttrib object. This is because there are two different types of XML formats, namely "XML elements" and "XML attributes". In the current implementation there is one affiliate networks: TradeTracker, which uses XML attributes.

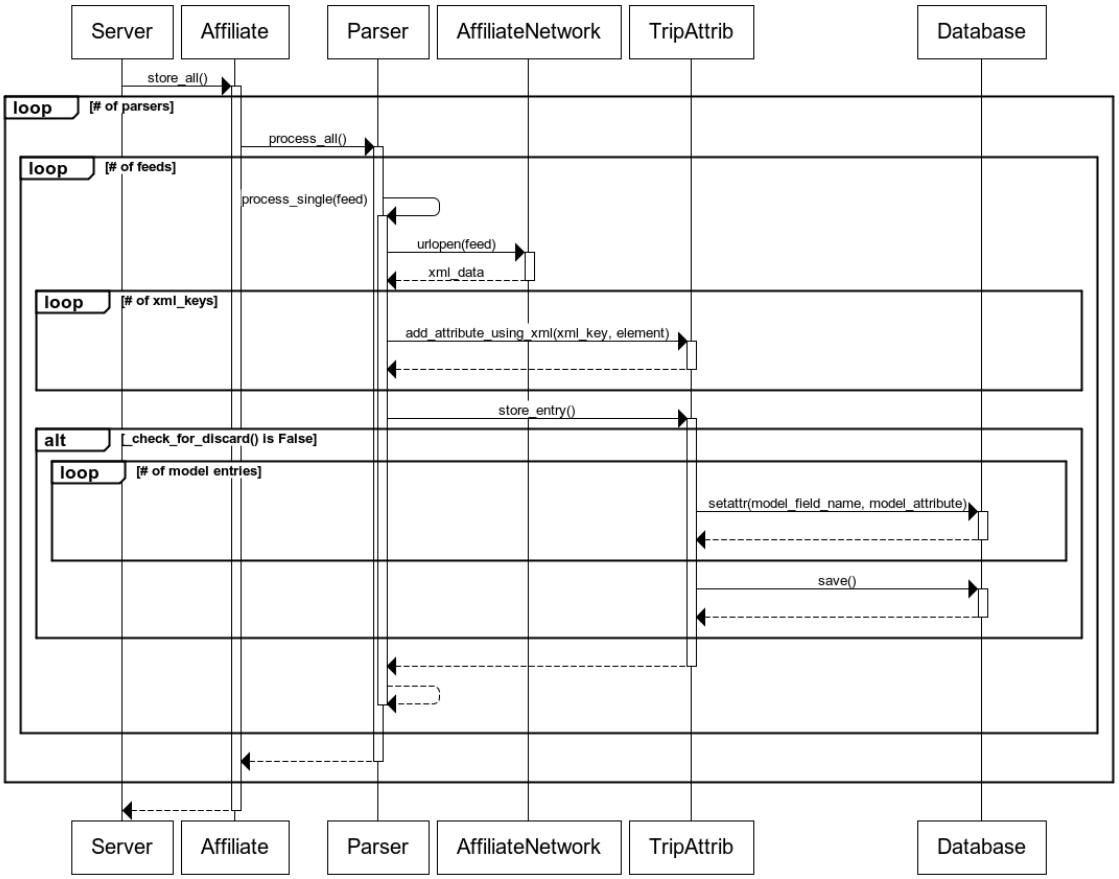

Figure 2.17: affiliate retrieve and save feed

Create a vacation

To create a vacation, the user has to fill in vacation data. This vacation data consists of a start date and end date and the offsets of these dates in integers, a budget per person and the amount of children and adults. When the user has an authentication token, the GUI sends the vacation data and the stored authentication token to the server, which redirects this data to the database for validation. If the authentication token is valid, the database returns the vac_id to the server which sends this data to the GUI. If the authentication token is invalid, the database returns a "not found" message which the server redirects as an error to the GUI.

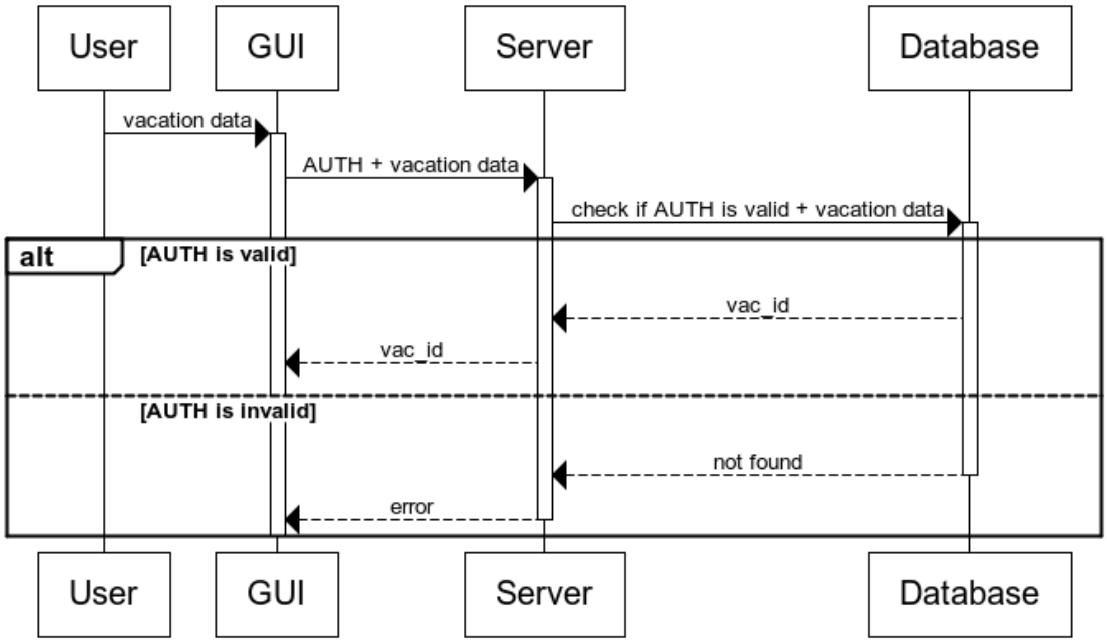

Figure 2.18: Create a vacation

Get vacation details

The retrieval of vacation details starts by users picking the vacation details they want to load. This request is send to the GUI and in order the GUI sends the authentication token and the vac_id to the server, upon which the server sends the authentication token and vac_id to verify their validity with the database. If the authentication token and vac_id are valid, the database can return the vacation data to the server, upon which the server sends it to the GUI. If the authentication token or vac_id are invalid, a "not found" response is sent from the database to the server. The server, in turn, returns an error to the GUI. In the end the GUI shows the result, either an error or the retrieved vacation details.

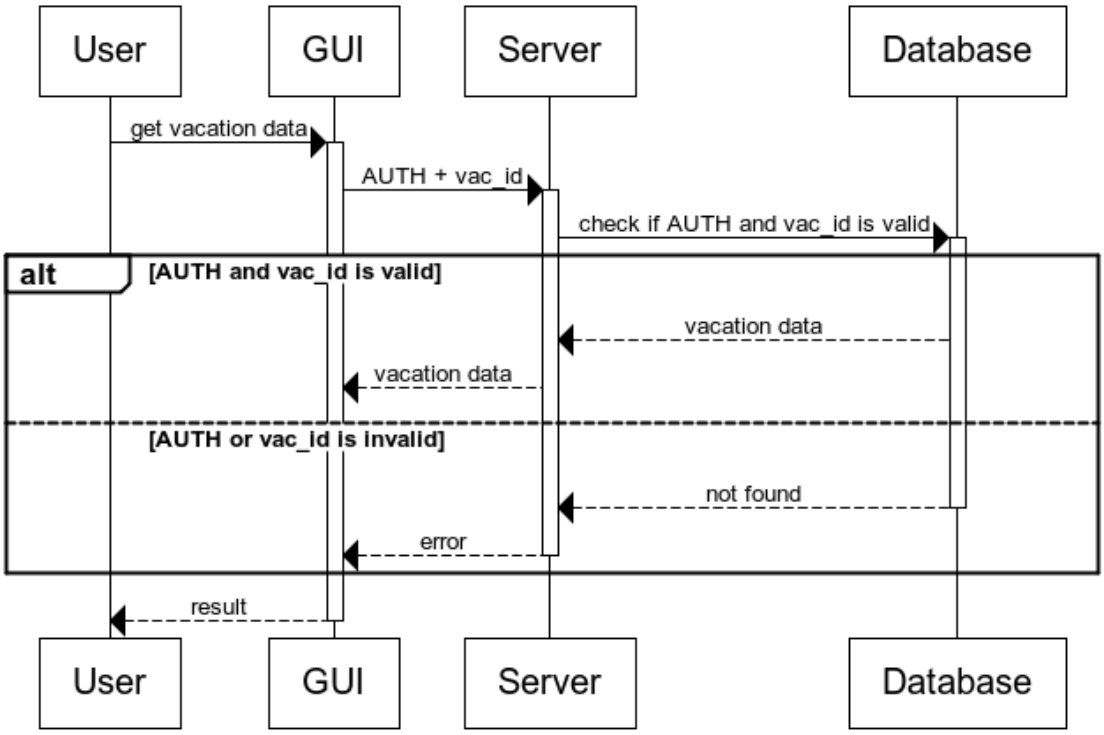

Figure 2.19: Get vacation details

Modify vacation details

If the user wants to modify their vacation details, the user changes the appropriate fields and clicks the submission button. This request is send to the GUI, which redirects the changed vacation data together with the authentication token and vac_id to the server. This is further sent to the database to verify the validity of the authentication token and the vac_id. If both are valid, a success is send from the database to the server, which the server redirects to the GUI.

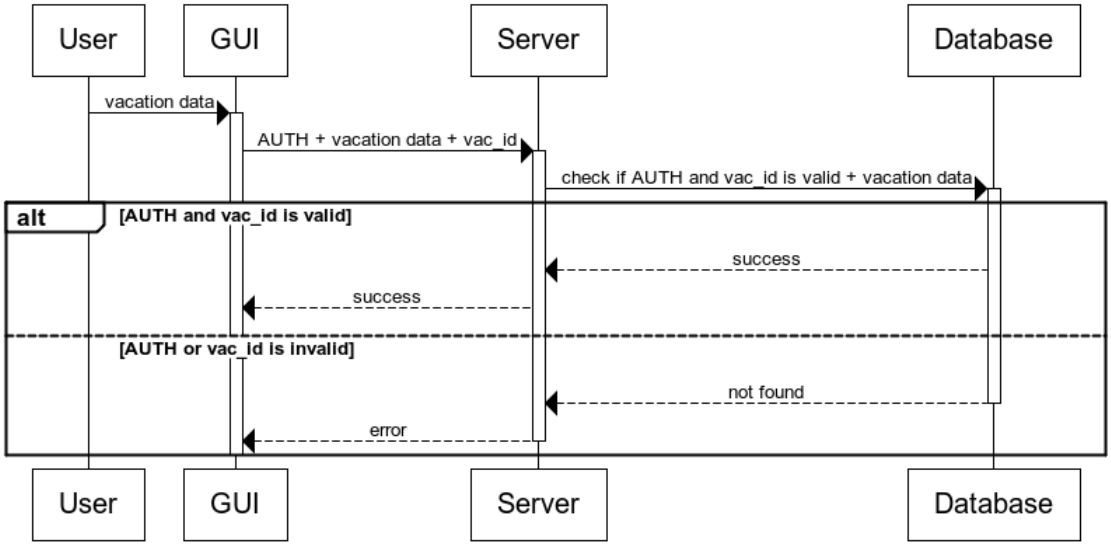

Figure 2.20: Modify vacation details

Getting images

If the user starts the interest analysis, images to be judged are required. If the user has sent their vacation details, the interest analysis can start. So whenever there are not enough images in the buffer a request to the server needs to be done. The GUI sends the authentication token, vacation id and the number of images needed n to the server. The server sends this data to the database to check the validity. If the information entered is valid n image references are returned to the server, upon which the server sends this data to the GUI. The GUI will then pick matching images sizes and send the request for these images to the server. The server will then respond with the correct sizes to the GUI. If any of the given data is invalid the database sends a "not found" response to the server and the server forwards this error to the GUI.

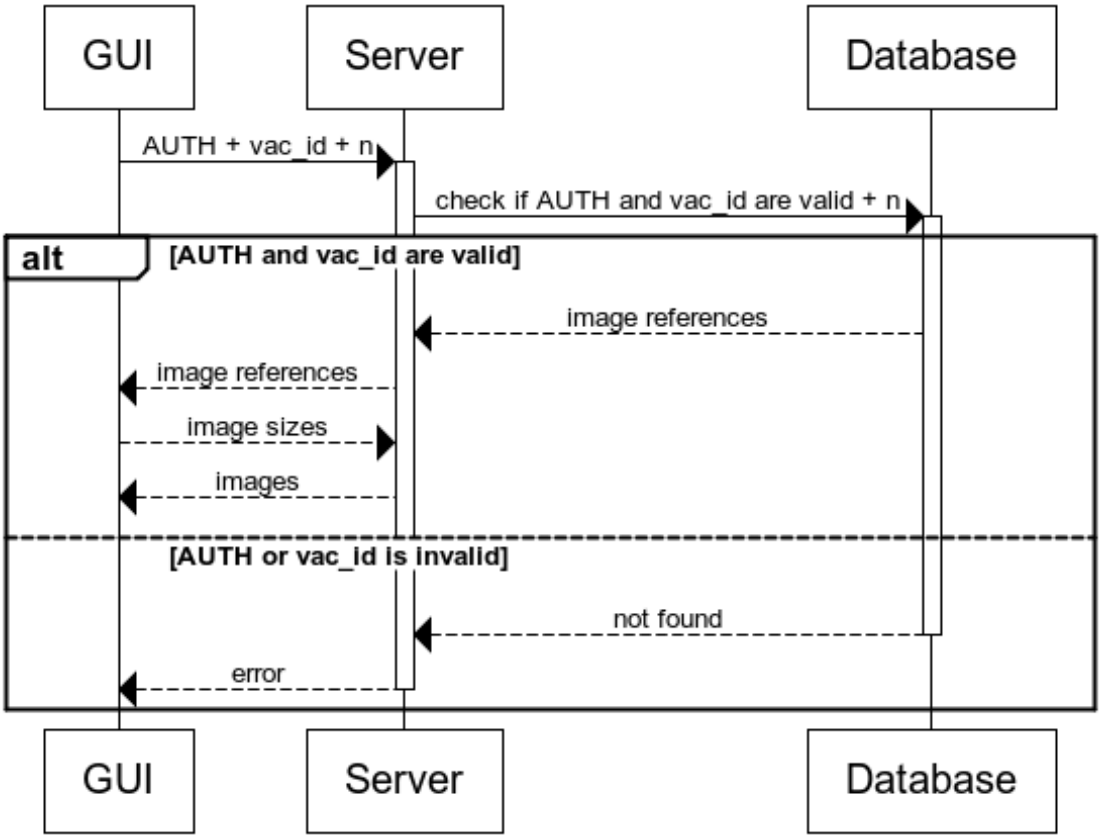

Figure 2.21: Retrieve images

Record a like or dislike to the back end

Recording a like or dislike to the back end starts with a (dis)like by the user of an image. By either clicking the buttons or swiping the images the user judges the image. This choice is recorded by the GUI and its result is sent to the server with all the necessary data for the database to store the choice. The server then sends this data to the database to check if the data for authentication vac_id and img_id is valid. If this data is valid, the Boolean like is recorded in the database and an image can be returned to the server. The server then returns another image to the GUI for the GUI to store in the buffer, to later show to the user. If any of the input data is invalid, the entry cannot be found in the database and this will be communicated to the server, which sends the appropriate error to the GUI.

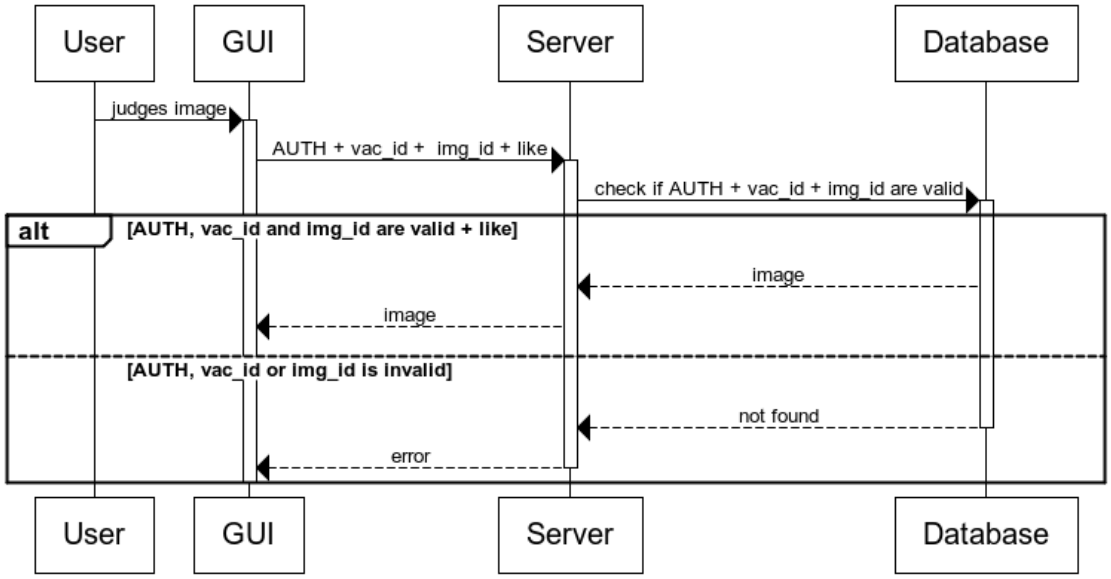

Figure 2.22: Record a like or dislike to the back end

Get recommendation

If the user is done judging images, a recommendation can be made on where the user should go on vacation. The first step for this is the user requesting a recommendation after judging a set amount of images. This request is send to the GUI, so the GUI can send the authentication token, n and the vacation ID to the server. The server in turns sends this data to the database to be validated. If the authentication token and vacation id are valid and also enough image judgments are recorded in the database, recommendations can be made. The database sends the made recommendations as locations and trips to the server which forwards them to the GUI. If the authentication token or vacation id is invalid, or not enough image judgments are recorded, an error is send from the database to the server, upon which the server sends this error to the GUI. Finally, the GUI sends the result to the user.

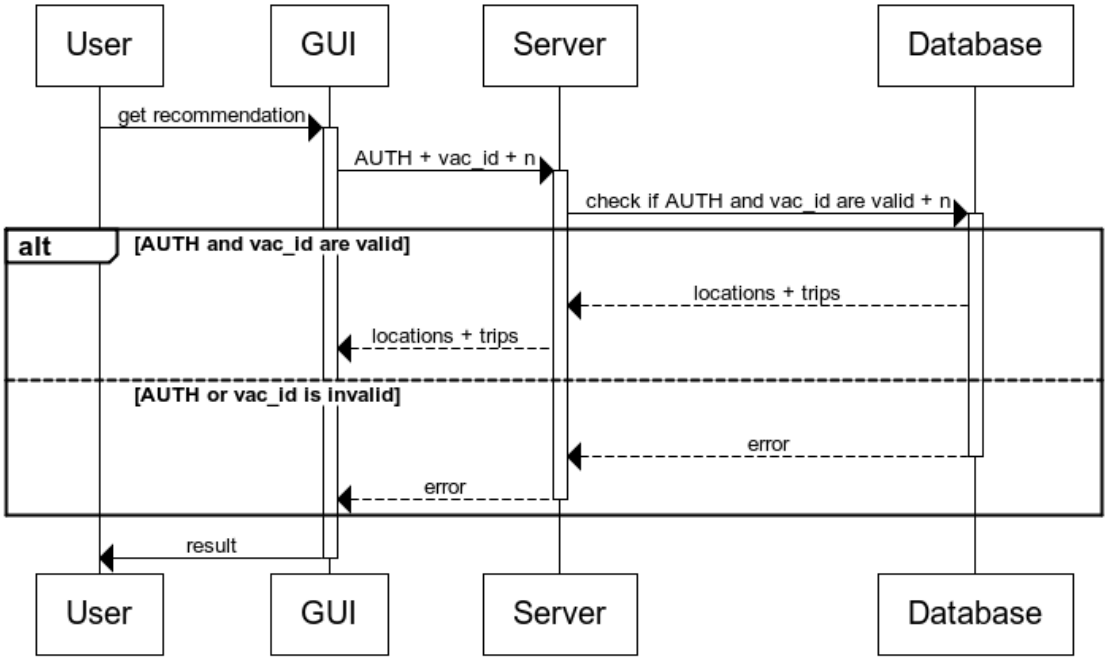

Figure 2.23: Get a recommendation after judgment

Get trips

The retrieval of trips is initiated by the user; the user requests more trips from the GUI. The GUI forwards this request with the authentication token and the location ID to the server. The server forwards this to the database to check the validity of this data. If the authentication token and the location ID are valid, the database can send more trips to the server which sends them to the GUI. If the authentication or location ID are invalid, the database sends an error to the server which forwards this to the GUI. In the end the GUI shows the result of this request.

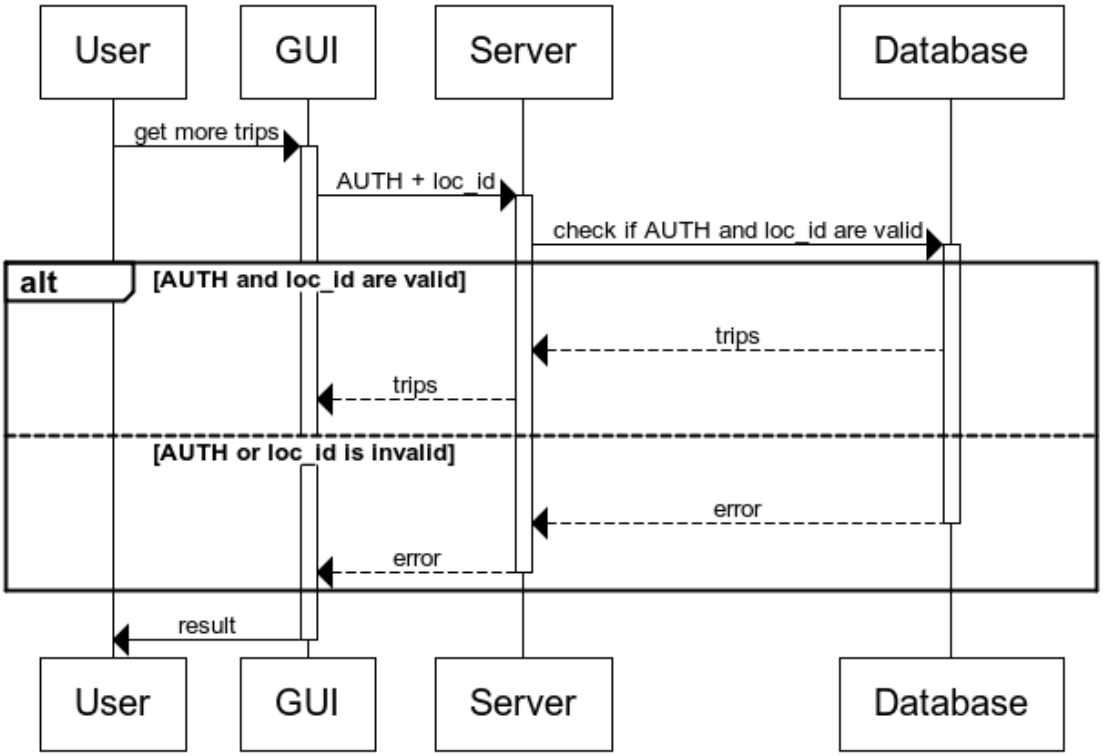

Figure 2.24: Get more trips

Save location overview

The saving of the location overview is initiated by the user; the user likes the recommendation and wants to, later on, go back to book a trip or find more trips. This request is sent to the server with the authentication token, location ID and an array of the trip IDs. The server forwards this data to the database to check the validity of the sent data. If this data is valid, the database responds with success to the server which forwards this message to the GUI. If the data is invalid, the database responds with an error to the server which forwards this to the GUI. Finally, the user gets feedback on the saving.

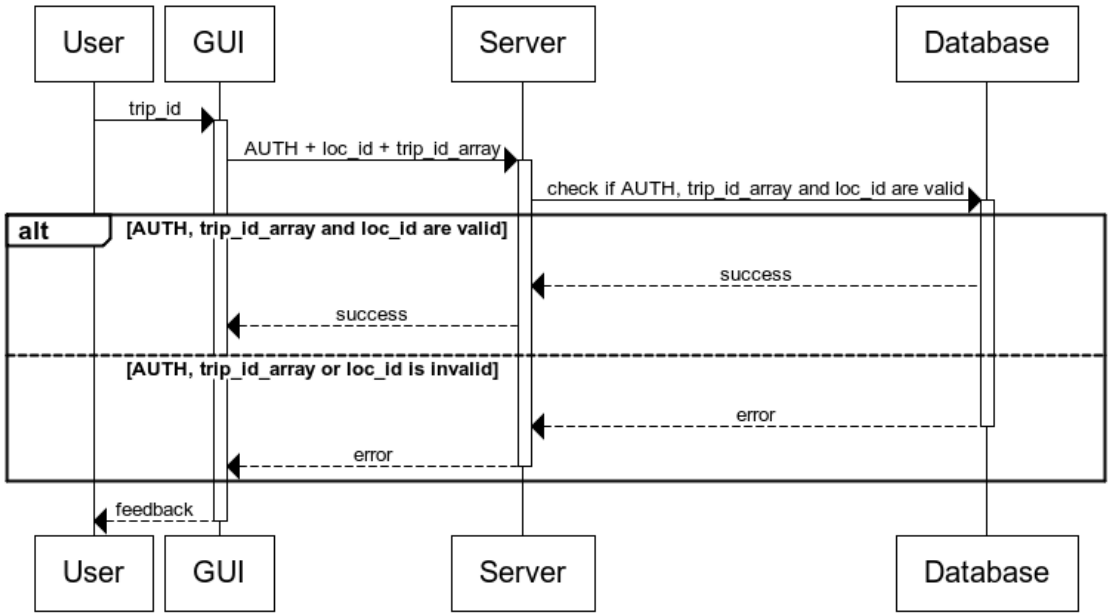

Figure 2.25: Save location

Load location overview

The user can load the saved location overviews; this request is initiated by the user, who returns to the recommended location. The request to load the location overview is send to the GUI, where the GUI adds the authentication token and the location ID. The server forwards the authentication token and the location ID to the Database to verify the validity. If the authentication token and location ID are valid, the database returns the stored trips to the server; in turn, the server sends this data to the GUI. If the authentication token or location ID are invalid, the database returns an error to the server, which redirects it to the GUI. Finally, the GUI shows the results to the user.

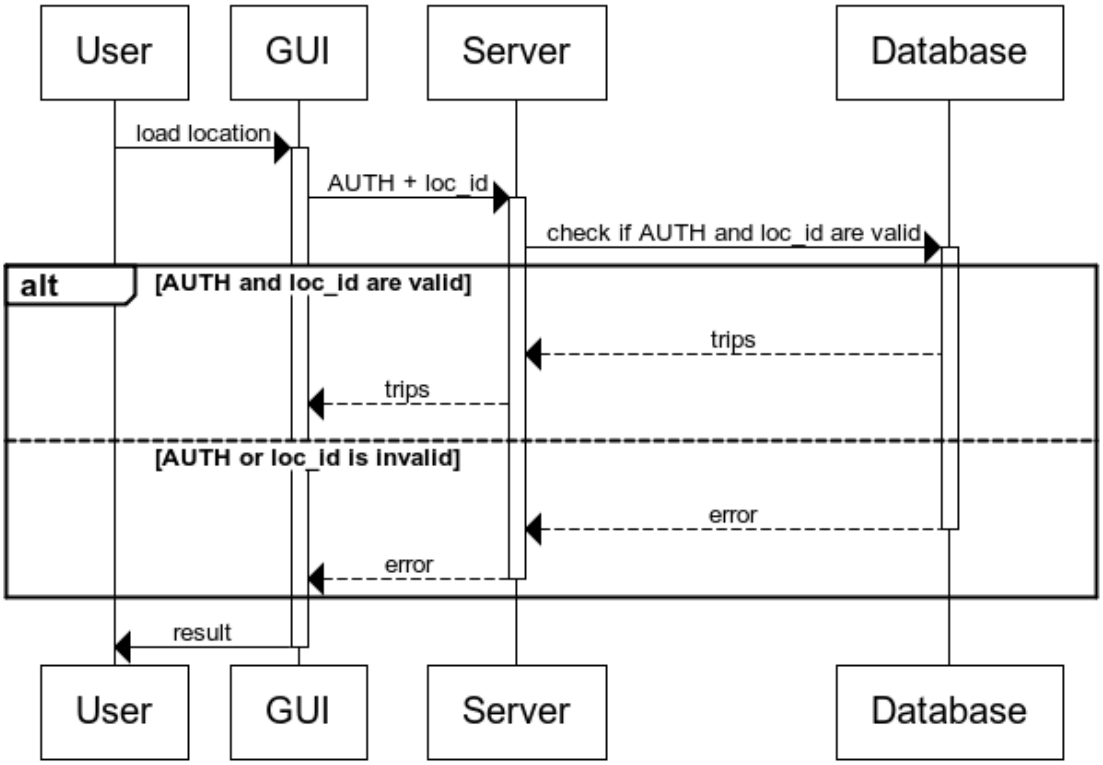

Figure 2.26: Load location

Delete location

The user can delete a recommended location by sending a request to the GUI. The GUI sends this requests with an authentication token and location ID to the server. The server sends the authentication token and location ID to the database to check the validity. If they are both valid, then the server receives a success message, which is forwarded to the GUI. If the authentication token or location ID is invalid, then the server receives an error message, which is forwarded to the GUI. Either way, the user does not see the recommended location as a saved location anymore.

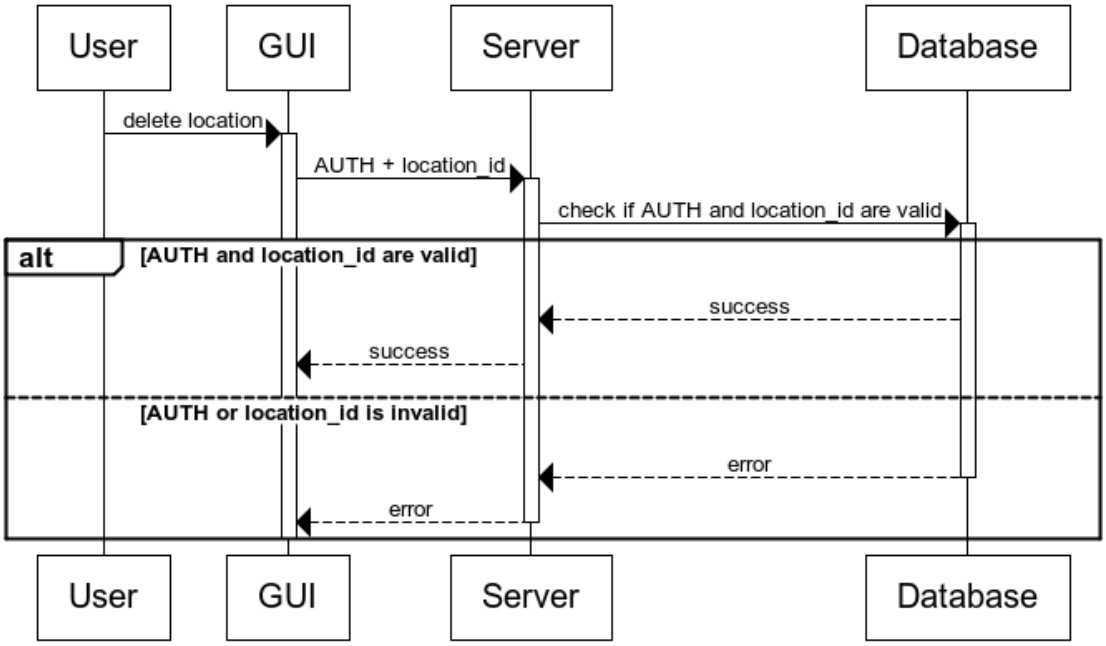

Figure 2.27: Delete location

Chapter 3
Specific requirements

The specific requirements discussed in this sections are divided into logical subsections. Important to mention is that there is one subsection for each class of specific requirements. To prioritize how important these requirements are, we use the MosCoW model [18]. The capital letters in MoSCoW stand for:

M *Must have*; these requirements are essential for the product.

S *Should have*; these requirements are not critical for the product to work, but are nearly as important as the must haves, meaning they must be implemented if at all possible.

C *Could have*; requirements which are not critical to the products success. If they can be implemented with little development costs, they can increase the Clients satisfaction.

W *Won't have*; these requirements will not be implemented in this project. However, it would be nice to have them in future versions of the product.

The priority for each requirement is listed with the respective requirement.

3.1 Functional requirements

The functional requirements are grouped by functions of the application and the back end. When there is interaction between TravelMatch app and back end those functions are described together. These groupings will give duplicates of some variable names, for example **SR2** and **SR16**. Although those requirements look the same, they still differ in the fact that they belong to different functions. This is also why **SR2** and **SR16** do not match to the same user requirements in the traceability matrix in Section 4.2.

3.1.1 Description requirements

The functional requirements are grouped with function in the TravelMatch app and back end response, as shown below:
Header about action in TravelMatch app or email provider
SR1 Function name and description, if it is a function in the TravelMatch app.

SR2-5 parameters to the function above.

The back end will respond with a status code; below all status codes are given.

1. **200:** OK
2. **201:** Created
3. **202:** Accepted

4. **204:** No Content
5. **400:** Bad Request

When the back end replies with a **200**, the following is returned:
SR6-8 All returned values from the back end, if more data than just a success message was returned.

Precondition and Postcondition for the TravelMatch app function and back end response handling.

3.1.2 User

Email registration

SR1 *Must have*
register function in the TravelMatch app to register the user with the parameters below via a call to the back end.

SR2 *Must have*
email :string
A string containing the e-mail address of the user.

SR3 *Must have*
password :string
A string containing the password of the user.

SR4 *Must have*
password2 :string
A string containing the a repeat of the user's password to check for mistakes, not sent to the back end.

The back end can then respond in several ways:

SR5 *Must have*
statusCode :integer
An integer returning the back end result; this may be one of the following replies:

- **201:** The user was created and stored in the database.
- **409:** The user already exists.

Precondition: password == password2 and email is a valid email address
Postcondition: An e-mail has been sent to the e-mail address given above; the user e-mail and password are stored, where the password is first hashed and salted.

Sending the verification e-mail

When an email registration is submitted, an e-mail is sent by Django through Mailgun to the supplied e-mail address with a link to verify the user's e-mail address. This happens outside the TravelMatch app.
The mail function is called with the following attributes:
SR6 *Must have*
to :string
The e-mail address to which the confirmation e-mail is sent.

SR7 *Must have*
userid :integer
The user ID generated for that user.

SR8 *Must have*
key :string
The randomly generated verification key.

Precondition: The user account has been created in the database with a valid e-mail address.
Postcondition: An e-mail has been sent to the e-mail address specified by the user.

Verify a users email address

It is possible to verify a e-mail address by visiting the link sent to the e-mail address that was entered during registration. This is outside of the TravelMatch app.

SR9 *Must have*
user_id :string
A string containing the user ID of the user.

SR10 *Must have*
key :string
A string containing the key send to the e-mail address of a registered user.

The back end can then respond in several ways:

SR11 *Must have*
statusCode :integer
An integer returning the back end result; this may be one of the following replies:

- **200:** The user was authenticated.
- **403:** The key is invalid.
- **404:** The user could not be found.

When the back end replies with a **200**, the following is returned:

SR12 *Must have*
content :string
An HTML page that congratulates the user.

When the back end replies with a **403** (the key is invalid), the following is returned:

SR13 *Must have*
content :string
An HTML page that lets the user send a new key to their mailbox.

When the back end replies with a **404** (user not found), the following is returned:

SR14 *Must have*
content :string
An HTML page that says that the user account does not exist and the user should make a new account in the app.

Precondition: The user is marked as not activated in the database and a confirmation e-mail has successfully been sent.
Postcondition: After successful verification, that same user is now marked as activated in the database.

Email login

SR15 *Must have*
login function in the TravelMatch app to login the User with the following parameters via a call to the back end

SR16 *Must have*
email :string
A string containing the email address of the user.

SR17 *Must have*
password :string
A string containing the password of the user.

The back end can then respond in several ways:

SR18 *Must have*
statusCode :integer
An integer returning the back end result; this may be one of the following replies:

- **204:** The user was logged in.
- **401:** Wrong credentials supplied for login.
- **403:** The account is not activated yet.
- **404:** The user could not be found.

Only when the back end replies with a **204**, the following is returned:
SR19 *Must have*
Authentication token :string
A unique string containing the authentication token of the user, encoded as a JWT token.

Precondition: email matches e-mail in database and the hashed password matches the hash of the password that was entered. The e-mail adress is verified, and thus authenticated.
Postcondition: An authentication token is returned to the TravelMatch app.

Facebook authentication

SR20 *Must have*
fbLogin function in the TravelMatch app to log in via Facebook authentication and also make a call to the back end to register.

SR21 *Must have*
fbid :string
A string containing the Facebook ID of the user given by Facebook.

SR22 *Must have*
fbtoken :string
A string containing the Facebook token of the user given by Facebook.

The back end can then respond in several ways:

SR23 *Must have*
statusCode :integer
An integer returning the back end result; this may be one of the following replies:

- **201:** The user was created and stored in the database.
- **202:** The user was already in the database.
- **403:** Facebook authentication failed.

When the back end replies with a **201** or **202**, the following is returned:
SR24 *Must have*
Authentication token :string
A unique string containing the authentication token of the user, encoded as a JWT token.

Precondition: The user successfully authenticates with Facebook
Postcondition: An authentication token is returned to the TravelMatch app

Get user details for user

SR25 *Must have*
getUserDetails function to retrieve the user details of the user with the authentication token stored in the TravelMatch app, via a back end call.

SR26 *Must have*
Authentication token :string
A string containing the authentication token of the user.

The back end can then respond in several ways:

SR27 *Must have*
statusCode :integer
An integer returning the back end result, this may be one of the following replies:

- **201:** The user was logged in.
- **403:** The user is not authenticated.

Only when the back end replies with a **201** (user logged in), the following is returned:
SR28 *Must have*
name :string
A string containing the name of the user

SR29 *Must have*
gender :string
A string containing the gender of the user

SR30 *Must have*
birth day :dateTime
A datetime object containing the birth date of the user

Precondition: The authentication token is available in the database
Postcondition: User details are returned.

Update user details for user

SR31 *Must have*
saveUserDetails function of the user in the TravelMatch app to change the user details of the currently logged in user identified by the authentication token. This is done via an call to the back end with the parameters below.

SR32 *Must have*
authentication token :string
A string containing the authentication token of the user that is logged in.

SR33 *Must have*
name :string
A string containing the name of the user

SR34 *Must have*
gender :string
A string containing the gender of the user

SR35 *Must have*
birthday :dateTime
A datetime object containing the birth date of the user

The back end can then respond in several ways:

SR36 *Must have*
statusCode :integer
An integer returning the back end result; this may be one of the following replies:

- **204:** The user data was updated.
- **412:** The user data is not saved because it was misformatted.

Precondition: The authentication token is available in the database
Postcondition: In the database the fields for name, gender and birthday are updated if they are non-empty and valid.

Delete user account

SR37 *Won't have*
deleteUser function of the User in the TravelMatch app to delete a user account

SR38 *Won't have*
Authentication token :string
A string containing the authentication token of the user.

SR39 *Won't have*
Identity :string
A string containing the identifying component of the user, only needed for the TravelMatch app.

The back end can then respond in several ways:

SR40 *Won't have*
statusCode :integer
An integer returning the back end result; this may be one of the following replies:

- **201:** The user was logged in.
- **404:** The user was not in the database.

Precondition: The authentication token and an identifier is given by the TravelMatch app
Postcondition: There is no entry for the user in the database or the TravelMatch app.

Login for a guest user

SR41 *Should have*
loginGuest function to login the guest account.

SR42 *Should have*
device_id :integer
An integer containing the device ID of the user

SR43 *Should have*
statusCode :integer
An integer returning the back end result; this may be one of the following replies:

- **200:** The guest account was logged in.
- **201:** The guest account was created and logged in.
- **409:** A conflict occurred.

Only when the back end replies with a **200** or **201**, the following is returned:
SR44 *Must have*
Authentication token :string
A unique string containing the authentication token of the user, encoded as an JWT token.

Precondition: The guest user has a device ID.
Postcondition: An authentication token is returned to the TravelMatch app for the guest user to use the application.

Delete guest account

SR45 *Should have*
deleteGuest function to delete the guest account.

SR46 *Should have*
device_id :integer
An integer containing the device ID of the user

SR47 *Should have*
statusCode :integer
An integer returning the back end result; this may be one of the following replies:

- **200:** The guest account was deleted.
- **404:** The guest account could not be found.

Precondition: The user has a device, and a guest account was made
Postcondition: The guest user's device ID is not stored in the database anymore. The authentication token in the TravelMatch app is also removed.

Check authentication

SR48 *Could have*
isAuthenticated function that will check if the user is authenticated.

SR49 *Could have*
AUTH :string
A string containing the authentication token of the user.

3.1.3 Vacation

Create or update vacation details for a user

SR50 *Must have*
create function of the vacation to create vacation details in the TravelMatch app via a call to the back end. The following parameters are used; the authentication token is used from the logged in user.

SR51 *Must have*
Authentication token :string
A string containing the authentication token of the user.

SR52 *Could have*
vac_name :string
A string containing the vacation name.

SR53 *Must have*
start_date :dateTime
A datetime object containing the starting date of the vacation.

SR54 *Must have*
start_date_extend :integer
An integer containing the amount of days the vacation can differ from the start date.

SR55 *Must have*
end_date :dateTime
A datetime object containing the date of return.

SR56 *Must have*
end_date_extend :integer
An integer containing the amount of days the vacation can differ from the end date.

SR57 *Should have*
adults :integer
An integer containing the amount of adults planning to go on vacation.

SR58 *Should have*
children :integer
An integer containing the amount of children planning to go on vacation.

SR59 *Could have*
vac_id :string
An integer containing the vacation ID; an optional parameter only available with existing vacation details.

SR60 *Must have*
budget :integer
An integer containing the amount of money per person the vacation can maximally cost.

The back end can then respond in several ways:
SR61 *Must have*
statusCode :integer
An integer returning the back end result; this may be one of the following replies:

- **201:** The vacation was created and stored in the database.
- **409:** The vacation name already existed.

Only when the back end replies with a **201**, the following is returned:
SR62 *Must have*
vac_id :integer
An integer containing the unique identifier of the vacation.

Precondition: The authentication token is a valid token in the database. persons is an integer higher than 0. budget is an integer higher or equal to 0. vac_name is unique. start_date is earlier than end_date.
Postcondition: In the database a new entry is made for this vacation and the vac_id is generated.

Get vacation details for a user

SR63 *Must have*
get function to retrieve previously filled in vacation details in the TravelMatch app via a call to the

back end with the following parameters retrieved in the TravelMatch from the logged in user.

SR64 *Must have*
Authentication token :string
A string containing the authentication token of the user that is logged in.

SR65 *Must have*
vac_id :string
A string containing the vacation name.

The back end can then respond in several ways:
SR66 *Must have*
statusCode :integer
An integer returning the back end result; this may be one of the following replies:

- **200:** The vacation has been found.
- **404:** There is no vacation for this ID.

Only when the back end replies with a **200**, the following is returned:
SR67 *Must have*
vac_id :integer
An integer containing the unique identifier of the vacation.

SR68 *Could have*
vac_name :string
A string containing the vacation name.

SR69 *Must have*
start_date :dateTime
A datetime object containing the start date of the vacation.

SR70 *Must have*
start_date_extend :integer
An integer containing the amount of days the vacation can differ from the start date.

SR71 *Must have*
end_date :dateTime
A datetime object containing the date of return.

SR72 *Must have*
end_date_extend :integer
An integer containing the amount of days the vacation can differ from the end date

---

SR73 *Must have*
adults :integer
An integer containing the amount of adults planning to go on vacation

---

SR74 *Must have*
children :integer
An integer containing the amount of children planning to go on vacation

---

SR75 *Must have*
budget :integer

An integer containing the amount of money per person the vacation can maximally cost

---

Precondition: The authentication token is available in the database and there is a vacation matching the vac_id.
Postcondition: A valid vacation is returned where the start_date is earlier than the end_date, persons is larger than 0 and budget is equal or higher than 0.

Delete vacation details for a user

SR76 *Should have*
delete function to delete the vacation details in the TravelMatch app, via a call to the back end with the following parameters. The TravelMatch app will provide the authentication token of the logged in user.

SR77 *Should have*
Authentication token :string
A string containing the authentication token of the user.

SR78 *Should have*
vac_id :integer
An integer containing the unique identifier of the vacation.

The back end can then respond in several ways:
SR79 *Should have*
statusCode :integer
An integer returning the back end result, this may be one of the following replies:

- **200:** The vacation was deleted.
- **404:** The vacation did not exist.

Precondition: The authentication token and vac_id is valid and available in the database.
Postcondition: The database entry for the vacation matching the vac_id is changed to deleted.

Get all vacation details for a user and the content

SR80 *Should have*
retrieveAll function to retrieve all vacation details and content in the TravelMatch app via a call to back end. The TravelMatch will handle the requirements.

It is possible to get all previously filled in vacation details from the back end. The back end will only need the AUTH parameter.
SR81 *Should have*
AUTH :string
A string containing the authentication token of the user.

The back end can then respond in several ways:
SR82 *Should have*
statusCode :integer
An integer returning the back end result, this may be one of the following replies:

- **200:** The vacation details have been found.
- **403:** The user is not authenticated.

Only when the back end replies with a **200**, the following is returned:

SR83 *Must have*

all_vac_details :array

An array containing all vacations made with their details inside.

SR84 *Must have*

start_date :dateTime

A datetime object containing the start date of the vacation, inside the all_vac_details array.

SR85 *Must have*

start_date_extend :integer

An integer containing the amount of days the vacation can differ from the start date, inside the all_vac_details array.

SR86 *Must have*

end_date :dateTime

A datetime object containing the date of return, inside the all_vac_details array.

SR87 *Must have*

end_date_extend :integer

An integer containing the amount of days the vacation can differ from the end date, inside the all_vac_details array.

SR88 *Must have*

adults :integer

An integer containing the amount of adults planning to go on vacation, inside the all_vac_details array.

SR89 *Must have*

children :integer

An integer containing the amount of children planning to go on vacation, inside the all_vac_details array.

SR90 *Could have*

vac_name :string

A string containing the vacation name, inside the all_vac_details array.

SR91 *Must have*

vac_id :integer

An integer containing the vacation ID, inside the all_vac_details array.

SR92 *Must have*

budget :integer

An integer containing the amount of money per person the vacation can maximally cost, inside the all_vac_details array.

Precondition: There is a valid authentication token matching the sent authentication token.
Postcondition: All stored vacation details are returned.

Get the next n image urls, with 1<=n<=100

SR93 *Must have*
obtain function in the TravelMatch app to retrieve the images needed for the interest analysis, via a call to the back end the images can be retrieved. The TravelMatch app will decide on the requirements for the call to the back end.

SR94 *Must have*
authentication token :string
A string containing the authentication token of the user that is stored.

SR95 *Must have*
vac_id :integer
An integer containing the identification of the vacation.

SR96 *Must have*
n :integer
An integer containing the amount of image URLs needed.

The back end can then respond in several ways:

SR97 *Must have*
statusCode :integer
An integer returning the back end result, this may be one of the following replies:

- **200:** The vacation details were found.
- **400:** The vac_id is not properly encoded.
- **404:** No vacation details for this vac_id could be found.

Only when the back end replies with a **200**, the following is returned:
SR98 *Must have*
image_array :array
An array containing the following items for each entry.

SR99 *Must have*
image_id :integer
An integer containing the unique identifier of the image inside the image_array.

SR100 *Must have*
sizes :array
An array containing the following items about the sizes of the image inside the image_array.

SR101 *Must have*
height :integer
An integer containing the height of an image inside the sizes array.

SR102 *Must have*
width :integer
An integer containing width of an image inside the sizes array.

SR103 *Must have*
url :string
A string containing the URL of the image inside the sizes array.

Precondition: The authentication token sent is a valid authentication token matching a entry in the database. The vac_id exists in the database. n is between or 1 and 100 inclusive.
Postcondition: The array size of image_array is equal to n.

Record a swipe to the back-end

SR104 *Must have*
record function to record the judgment of an image in the TravelMatch app to the back end via a call. The user will judge the image with a swipe or click as parameter choice and the TravelMatch app will do the rest of the requirements

---

SR105 *Must have*
AUTH :string
A string containing the authentication token of the user.

SR106 *Must have*
vac_id :integer
An integer containing the unique identifier of the vacation details.

SR107 *Must have*
img_id :integer
An integer containing the unique identifier of an image.

SR108 *Must have*
choice :boolean
A Boolean containing the judgment of the user of the image.

The back end can then respond in several ways:
SR109 *Must have*
statusCode :integer
An integer returning the back end result, this may be one of the following replies:

- **200:** The vacation details were found.
- **404:** No vacation details for this vac_id could be found.
- **405:** There is no active image for the provided image_id.

Only when the back end replies with a **200**, the following is returned:
SR110 *Must have*
empty :array
An array containing an empty object for consistency.

Precondition: The authentication token is a valid token matching a entry in the database. There is a vac_id matching the entry in the database. img_id exists in the database.
Postcondition: The database stores the recorded judgment.

Record a swipe to the back-end and request n images

SR111 *Must have*
record function to record the judgment of an image in the TravelMatch app and request more images

to the back end via a call. The user will judge the image with a swipe or click as parameter choice and the TravelMatch app will do the rest of the requirements

---

SR112 *Must have*
AUTH :string
A string containing the authentication token of the user.

SR113 *Must have*
vac_id :integer
An integer containing the unique identifier of the vacation details.

SR114 *Must have*
img_id :integer
An integer containing the unique identifier of an image.

SR115 *Must have*
choice :boolean
A Boolean containing the judgment of the user of the image.

SR116 *Must have*
n :integer
An integer containing the amount of images requested; depends on the progress and buffer.

The back end can then respond in several ways:
SR117 *Must have*
statusCode :integer
An integer returning the back end result, this may be one of the following replies:

- **200:** The vacation details were found.
- **404:** No vacation details for this vac_id could be found.
- **405:** There is no active image for the provided image_id.

Only when the back end replies with a **200**, the following is returned:
SR118 *Must have*
image_array :array
An array containing the following items for each entry.

SR119 *Must have*
image_id :integer
An integer containing the unique identifier of the image inside the image_array.

SR120 *Must have*
sizes :array
An array containing the following items about the sizes of the image inside the image_array.

SR121 *Must have*
height :integer
An integer containing the height of an image inside the sizes array.

SR122 *Must have*
width :integer
An integer containing width of an image inside the sizes array.

SR123 *Must have*
url :string
A string containing the URL of the image inside the sizes array.

Precondition: The authentication token is a valid token which matches an entry in the database. vac_id and img_id exist in the database. The integer n is between 1 and 100 inclusive.
Postcondition: the returned image_array size is equal to the sent integer n.

Get recommendation

SR124 *Must have*
getRecommendation function to retrieve the recommendation based on the Travel DNA via a call to the back end. The TravelMatch app will supply the requirements for the call to the back end.

SR125 *Must have*
AUTH :string
A string containing the authentication token of the user.

SR126 *Must have*
vac_id :integer
An integer containing the unique identifier of the vacation details.

SR127 *Must have*
n :integer
An integer containing the amount of recommendations requested.

The back end can then respond in several ways:

SR128 *Must have*
statusCode :integer
An integer returning the back end result, this may be one of the following replies:

- **200:** A recommendation was generated.
- **404:** No vacation details for this vac_id could be found.
- **412:** The request is not ready, because too little images have been judged.

Only when the back end replies with a **200**, the following is returned:
SR129 *Must have*
recommendation :array
An array containing an location object for the location fitting tot the Travel DNA.

SR130 *Must have*
location :dictionary
A dictionary containing an location object for the location fitting tot the Travel DNA.

SR131 *Must have*
loc_id :integer
An integer containing an unique identifier for the location, inside the location dictionary.

SR132 *Must have*
loc_city :string
A string containing the city of the location, inside the location dictionary.

SR133 *Must have*
loc_region :string
A string containing the region of the location, inside the location dictionary.

SR134 *Must have*
loc_country :string
A string containing the country of the location inside the location dictionary.

SR135 *Must have*
trips :array
An array containing trip objects.

SR136 *Must have*
trip_id :integer
A integer containing the unique identifier of the trip inside the trips array.

SR137 *Must have*
trip_name :string
A string containing the name of the trip inside the trips array.

SR138 *Must have*
trip_description :string
A string containing the description of the trip inside the trips array.

SR139 *Must have*
trip_price :integer
An integer containing the price of the trip inside the trips array.

SR140 *Must have*
trip_UserRating :integer
An integer containing the user rating of the trip inside the trips array.

SR141 *Must have*
trip_HotelRating :integer
An integer containing the Hotelstars rating of the trip inside the trips array.

SR142 *Must have*
trip_link :string
A string containg the link to book the trip, inside the trips array.

SR143 *Must have*
trip_image An image about the trip, inside the trips array.

Precondition: The authentication token sent is matched with a entry in the database. The vac_id is an existing entry in the database. The user has judged a set amount of images. The integer n is between 1 and 2 inclusive.
Postcondition: Recommendations are send to the user that fit the Travel DNA.

Refresh a hotel overview

SR144 *Should have*
retrieveTrips function to retrieve more trips for the made recommendation. The TravelMatch app will supply the requirements for the call to the back end.

SR145 *Should have*
AUTH :string
A string containing the authentication token of the user.

SR146 *Should have*
loc_id :integer
An integer containing the unique identifier of the location.

The back end can then respond in several ways:

SR147 *Should have*
statusCode :integer
An integer returning the back end result, this may be one of the following replies:

- **200:** Trips were found.
- **404:** The location could not be found.

Only when the back end replies with a **200**, the following is returned:
SR148 *Should have*
trips :array
An array containing trip objects.

SR149 *Should have*
trip_id :integer
A integer containing the unique identifier of the trip inside the trips array.

SR150 *Should have*
trip_name :string
A string containing the name of the trip inside the trips array.

SR151 *Should have*
trip_description :string
A string containing the description of the trip inside the trips array.

SR152 *Should have*
trip_price :integer
An integer containing the price of the trip inside the trips array.

SR153 *Should have*
trip_UserRating :integer
An integer containing the user rating of the trip inside the trips array.

SR154 *Should have*
trip_HotelRating :integer
An integer containing the Hotelstars rating of the trip inside the trips array.

SR155 *Should have*
trip_link :string

A string containg the link to book the trip, inside the trips array.

SR156 *Should have*
trip_image An image about the trip, inside the trips array.

Precondition: The authentication token sent is matched with a entry in the database. The loc_id is an existing entry in the database.
Postcondition: Trips for the locations are sent to the TravelMatch app.

Save a location

SR157 *Should have*
saveLocation function to save the location that was recommended in the TravelMatch app. The TravelMatch app will supply the requirements for the call to the back end.

SR158 *Should have*
AUTH :string
A string containing the authentication token of the user.

SR159 *Should have*
loc_id :integer
An integer containing the unique identifier of the location.

SR160 *Should have*
trip_id_array :array
An array containing the unique identifiers of the trips stored.

The back end can then respond in several ways:

SR161 *Must have*
statusCode :integer
An integer returning the back end result, this may be one of the following replies:

- **200:** The location was saved.
- **404:** The location could not be found.

Precondition: The authentication token sent is matched with a entry in the database. The loc_id and trip_id are existing entries in the database.
Postcondition: In the database, an entry is made to store the location; a previous entry for that user is removed.

Load a location

SR162 *Should have*
loadLocation function to load the trips that was recommended in the TravelMatch app. The TravelMatch app will supply the requirements for the call to the back end.

SR163 *Should have*
AUTH :string
A string containing the authentication token of the user.

SR164 *Should have*
loc_id :integer
An integer containing the unique identifier of the location.

The back end can then respond in several ways:

SR165 *Must have*
statusCode :integer
An integer returning the back end result, this may be one of the following replies:

- **200:** The location was found.
- **404:** The saved location could not be found.

Only when the back end replies with a **200**, the following is returned:
SR166 *Must have*
trips :array
An array containing trip objects.

SR167 *Must have*
trip_id :integer
A integer containing the unique identifier of the trip inside the trips array.

SR168 *Must have*
trip_name :string
A string containing the name of the trip inside the trips array.

SR169 *Must have*
trip_description :string
A string containing the description of the trip inside the trips array.

SR170 *Must have*
trip_price :integer
An integer containing the price of the trip inside the trips array.

SR171 *Must have*
trip_UserRating :integer
An integer containing the user rating of the trip inside the trips array.

SR172 *Must have*
trip_HotelRating :integer
An integer containing the Hotelstars rating of the trip inside the trips array.

SR173 *Must have*
trip_link :string
A string containing the link to book the trip, inside the trips array.

SR174 *Must have*
trip_image An image about the trip, inside the trips array.

Precondition: The authentication token sent is matched with a entry in the database. The loc_id is an existing entry in the database with saved trips matched to it.
Postcondition: The database returns found each trip that was found, if it is still available via the

affiliate network. The trips are displayed in the TravelMatch app

---

Delete a location

SR175 *Could have*
deleteLocation function to remove the location that was recommended in the TravelMatch app. The TravelMatch app will supply the requirements for the call to the back end.

SR176 *Could have*
AUTH :string
A string containing the authentication token of the user.

SR177 *Could have*
loc_id :integer
An integer containing the unique identifier of the location.

The back end can then respond in several ways:

SR178 *Must have*
statusCode :integer
An integer returning the back end result, this may be one of the following replies:

- **200:** The location was deleted.
- **404:** The saved location could not be found.

Precondition: The authentication token sent is matched with a entry in the database. The loc_id is an existing entry in the database with saved trips matched to it.
Postcondition: In the database, the entry that stored the trip is removed.

3.1.4 CMS

SR179 *Must have*
In the CMS, it is possible to add images.

Precondition: The user is authenticated and provides an image.
Postcondition: In the database, the provided image is added with default tag values.

SR180 *Must have*
The CMS uses two versions, so changes are not directly deployed to the system.

Precondition: The user is authenticated and active version is selected.
Postcondition: Images with the active version are available to the database to be sent to the user.

SR181 *Must have*
In the CMS, locations can be added.

Precondition: The user is authenticated and a location is added.
Postcondition: In the database, a location is added with default tag attributes on not active version.

SR182 *Won't have*
In the CMS, it is possible to add a feed to the affiliate.

Precondition: The user is authenticated and a valid feed is added.
Postcondition: The server can retrieve data from the feed.

SR183 *Must have*
In the CMS, locations have a city field.

Precondition: The user is authenticated and sets the city for a location.
Postcondition: A location has a city attribute.

SR184 *Must have*
In the CMS, locations have a region field.

Precondition: The user is authenticated and sets the region for a location.
Postcondition: A location has a region attribute.

SR185 *Must have*
In the CMS, locations have a country field.

Precondition: The user is authenticated and sets the country for a location.
Postcondition: A location has a country attribute.

SR186 *Won't have*
In the CMS, a priority can be set for trips.

Precondition: The user is authenticated and sets valid priorities.
Postcondition: Priorities are set in the database.

SR187 *Won't have*
In the CMS, the set amount of images needed for the interest analysis can be set.

Precondition: The user is authenticated and a valid integer is set for images needed.
Postcondition: An integer is set for the amount of image needed for the interest analysis.

SR188 *Must have*
In the CMS, DNA attributes can be added.

Precondition: The user is authenticated and adds a name for the attribute.
Postcondition: In the database the DNA attribute is added and a default value is filled in for each relation.

CMS login

SR189 *Must have*
loginCMS function to login the CMS

SR190 *Must have*
username :string
A string containing the unique identifier for a user parameter to loginCMS.

SR191 *Must have*
password :string
A string containing the password for a user parameter to loginCMS.

Precondition: The user fills in correct username and password matching a entry in the database.
Postcondition: The user has access to the CMS.

Modify tag of an image

SR192 *Must have*
modifyImageTag function to modify a image tag

SR193 *Must have*
tag :integer
An integer containing a tag value for an image parameter to modifyImageTag

SR194 *Must have*
image_id :integer
An integer containing the identifier for the image; parameter to modifyImageTag

Precondition: the user is authenticated, a valid image is provided and a valid tag value between 0 and 100 inclusive is given.
Postcondition: The tag value is matched to the image and stored in the database.

Modify tag of a location

SR195 *Must have*
ModifyLocationTag function in the CMS to modify the locations tag value.

SR196 *Must have*
tag :integer
An integer containing the tag value; parameter to ModifyLocationTag

SR197 *Must have*
location_id :integer
An integer containing the identifier of the image; parameter to ModifyLocationTag

Precondition: The user is authenticated, provides a tag value between 0 and 100 inclusive and provides a valid location.
Postcondition: The tag value is matched to a location and stored in the database.

3.1.5 AI module

SR198 *Won't have*
The location tags are updated with each booked trip.

Precondition: An user has booked a trip via the TravelMatch app. The affiliate service gives feedback

on the booking that was made.
Postcondition: The location attributes are updated accordingly.

SR199 *Should have*
The authentication token expires after a week.

Precondition: Inside the token a date can be specified for expiration of the token.
Postcondition: The token can be read for the expiration date.

SR200 *Must have*
The API is able to uniquely identify users.

Precondition: An authentication token is unique and made as a JSON web token.
Postcondition: The authentication token is stored in the database and on the TravelMatch app if the user is authenticated.

SR201 *Should have*
The API requests should be logged.

Precondition: API requests have been made.
Postcondition: A file is present that records the API requests that were made.

SR202 *Must have*
Images are shown to the user to form a Travel DNA.

Precondition: Image choices are needed for the Travel DNA.
Postcondition: Information gain is optimized to calculate which image should be shown to the user to gain the most information for the Travel DNA.

SR203 *Must have*
The TravelMatch app hotel overview screen has two recommendations.

Precondition: A user has judged a set amount of images, creating a Travel DNA.
Postcondition: By the calculated similarity between Travel DNA and locations, two recommendations are picked of the highest similarity.

SR204 *Could have*
The TravelMatch app hotel overview screen can start a new interest analysis.

Precondition: The user has received recommendations and wants another recommendation.
Postcondition: The user is given another interest analysis sequence to receive a different recommendation, the previously created Travel DNA is included in this new interest analysis.

Recommendation calculation

SR205 *Must have*
Ranking function to rank Travel DNAs to locations in similarity

---

SR206 *Must have*
Travel DNA :array
An array containing all choices made by the user with tag values; parameter to Ranking.

SR207 *Must have*
locations :array
An array containing all locations with tag values; parameter to Ranking.

Precondition: Travel DNA and locations are vectors in a positive space based on their tag values.
Postcondition: Travel DNA and locations matching is based on cosine similarity. A ranking is made based on the most similarity to the Travel DNA.

3.1.6 Analytics

SR208 *Must have*
The TravelMatch app uses Angulartics.

Precondition: In every screen in the TravelMatch app Angulartics is implemented to monitor the activity.
Postcondition: Angulartics will record time stamps and all redirects in the TravelMatch app.

SR209 *Won't have*
Feedback from the analytics system is used to update location attributes.

Precondition: The user is using the TravelMatch app hotel overview screen extensively.
Postcondition: Location attributes are changed accordingly.

3.1.7 Affiliate network

Link Affiliate

SR210 *Should have*
linkAffiliate A function to link an affiliate service.

SR211 *Should have*
affiliate_url :string
A string containing the URL for a affiliate service; parameter to linkAffiliate

Precondition: A valid affiliate url has been given.
Postcondition: A link has been made to the provided affiliate service.

Retrieve data from feed

SR212 *Should have*
saveFeed An affiliate service can be used to save and parse feeds.

SR213 *Should have*
feed_url :string
A string containing the URL for a feed of an affiliate service, parameter to saveFeed

SR214 *Must have*
statusCode :integer
An integer returning the Affiliate service result; this may be one of the following replies:

- **200:** The feed is available.
- **401:** Not authorized.

Only when the affiliate service replies with a **200**, the following is returned:
SR215 *Should have*
feed_output :xmlData
xmlData containing trip offers from a feed.

Precondition: A valid URL of the feed has been chosen, which is connected to the respective affiliate service.
Postcondition: Data is retrieved from the feed.

Parse feed

SR216 *Should have*
parseFeed function to parse feed output to an entry in the database

SR217 *Should have*
feed_output :xmlData
xmlData from a feed.

Precondition: Feed ouput is in XML and the parser can understand the feed output.
Postcondition: Feed ouput is translated to an entry in the database and stored in the database.

3.2 Non-Functional requirements

3.2.1 Content Management System

SR218 *Must have*
The CMS is available in the English language.

SR219 *Must have*
The CMS is created using the Django framework, which uses Python.[24]

SR220 *Must have*
The CMS is available on Firefox 27.0 and later versions.

SR221 *Must have*
The CMS is available on Google Chrome 31.0 and later versions.

SR222 *Should have*
The CMS uses a MySQL database.

3.2.2 API

SR223 *Must have*
The API sends data in JSON format.

SR224 *Must have*
The API receives data in JSON format.

SR225 *Must have*
The API can handle multiple concurrent requests.

3.2.3 TravelMatch app

SR226 *Must have*
The TravelMatch app can run on smart-phone devices, running Android versions 4.1 "Jelly Bean" or newer.

SR227 *Must have*
The TravelMatch app can run on tablet devices devices, running Android versions 4.1 "Jelly Bean" or newer.

SR228 *Must have*
The TravelMatch app can run on smart-phone devices running iOS versions 7.0 or newer.

SR229 *Must have*
The TravelMatch app can run on tablet devices running iOS versions 7.0 or newer.

SR230 *Won't have*
The TravelMatch app is available for free through the Google Play Store for all supported Android devices.

SR231 *Won't have*
The TravelMatch app is available for free thought the App Store for all supported iOS devices.

SR232 *Won't have*
The TravelMatch app can run on Android Wear.

SR233 *Won't have*
The TravelMatch app can run on Android TV.

SR234 *Won't have*
The TravelMatch app can run on Apple TV.

SR235 *Won't have*
The TravelMatch app can run on Apple Watch.

SR236 *Won't have*
The TravelMatch app can run on Windows Phone.

SR237 *Won't have*
The TravelMatch app can run on desktop or laptop operating systems.

SR238 *Must have*
The TravelMatch app is made using the Ionic framework.[22]

SR239 *Must have*
Multiple user applications are able to connect to the API simultaneously.

SR240 *Must have*
The TravelMatch app is displayed in portrait mode on smartphones.

SR241 *Must have*
The TravelMatch app supports a sidebar.

SR242 *Must have*
The TravelMatch app supports a vacation details screen.

SR243 *Must have*
The TravelMatch app supports an interest analysis screen.

SR244 *Must have*
The TravelMatch app supports a user profile screen.

SR245 *Must have*
The TravelMatch app supports a log in screen.

SR246 *Must have*
The TravelMatch app supports a registration screen.

SR247 *Must have*
The TravelMatch app supports a hotel overview screen.

SR248 *Must have*
The TravelMatch app supports an "about" screen.

SR249 *Could have*
The TravelMatch app supports a saved destination.

SR250 *Must have*
The TravelMatch app supports a welcome screen.

SR251 *Must have*
The TravelMatch app stores the authentication token of a user.

SR252 *Must have*
The TravelMatch app stores the identifier of a user.

SR253 *Must have*
The TravelMatch app has a loading screen.

SR254 *Must have*
The TravelMatch app has a header.

SR255 *Must have*
The TravelMatch app supports a back button.

3.2.4 Analytics

SR256 *Won't have*
There is an analytics overview that shows which information has been recorded.

3.2.5 Adaptability

SR257 *Won't have*
In the back end, a Waverunner-like system can easily replace the affiliate network.

SR258 *Won't have*
The TravelMatch app supports booking functionality.

SR259 *Won't have*
The back end provides bookings.

SR260 *Must have*
The TravelMatch app supports adding authentication mechanisms.

3.2.6 Capability

SR261 *Must have*
The TravelMatch app will show TravelMatch logo when the application is started.

SR262 *Must have*
The TravelMatch app uses a language service to support different languages.

SR263 *Must have*
The database will store Travel DNAs for each user for each vacation.

3.2.7 Performance

SR264 *Could have*
It takes less than 0.25 seconds on any of the supported devices to display content when using the TravelMatch app.

SR265 *Should have*
It takes less than 1 seconds on any of the supported devices to display content when using the TravelMatch app.

SR266 *Must have*
It takes less than 2 seconds on any of the supported devices to display content when using the TravelMatch app.

SR267 *Should have*
It takes less than 5 to gain visual feedback from the CMS.

SR268 *Must have*
It takes less than 15 to gain visual feedback from the CMS.

SR269 *Should have*
It takes less than 2 to gain visual feedback from the recommendation screen.

SR270 *Must have*
It takes less than 5 to gain visual feedback from the recommendation screen.

SR271 *Must have*
The database can hold and supports at least 1000 images.

SR272 *Should have*
The database can hold and supports at least 10000 images.

SR273 *Must have*
The database can hold and supports at least 100 tags.

SR274 *Should have*
The database can hold and supports at least 500 tags.

SR275 *Must have*
The database can hold and supports at least 10000 guest users.

SR276 *Should have*
The database can hold and supports at least 500000 guest users.

SR277 *Must have*
The database can hold and supports at least 500 analytic event records per user.

SR278 *Must have*
The database can hold and supports at least 200 different destinations.

SR279 *Should have*
The database can hold and supports at least 1500 different destinations.

3.2.8 Licensing

SR280 *Should have*
TravelMatch is licensed under the terms of the MIT license.[37]

SR281 *Should have*
TravelMatch is licensed under the terms of the Apache license.[38]

Chapter 4
Requirements traceability matrix

Tracing software requirements to user requirements is vital in the software production process. It allows developers to relate specific parts of the software to the user requirements. In this way, each developer should know what the purpose of a certain component is.

4.1 User requirements to software requirements

This section describes all software requirements related to a certain user requirement. Hence, we can see which software requirements were made to fulfill a certain user requirement. This gives a nice method to check the software requirements. For each user requirement, we inspect the related software requirements and ask ourselves if the software requirements are sufficient to fulfill the user requirement. If not, the software requirements are probably incomplete. The Tractability matrix is shown in the following table.

UCR1	**SR255, SR262**
UCR2	**SR250, SR48, SR49**
UCR2a	**S250, SR41, SR42, SR43, SR44**
UCR2b	**SR250, SR245**
UCR2c	**SR250, SR246**
UCR2d	**SR250, SR20, SR21, SR22, SR23, SR24**
UCR3	**SR238**
UCR4	**SR238**
UCR5	**SR251, SR48, SR49**
UCR6	**SR253**
UCR7	**SR238**
UCR8	**SR240**
UCR9	**SR240**
UCR10	**SR240, SR238**
UCR11	**SR240, SR238**
UCR12	**SR263**
UCR13	**SR241, SR244**
UCR14	**SR241, SR162, SR163, SR164, SR165**
UCR15	**SR241, SR162, SR163, SR164, SR165, SR249**
UCR16	**SR241, SR175, SR176, SR177, SR178, SR249**
UCR17	**SR241, SR251**
UCR18	**SR241, SR248**
UCR19	**SR241, SR228, SR229**
UCR20	**SR241, SR252, SR251**
UCR21	**SR241, SR245**
UCR22	**SR241, SR246**
UCR23	**SR241, SR248**
UCR24	**SR246, SR1, SR2, SR3, SR4, SR5, SR6, SR7 SR8, SR9, SR10, SR11, SR12, SR13, SR14**
UCR25	**SR246, SR20, SR21, SR22, SR23, SR24**
UCR26	**SR246 , SR239**
UCR27	**SR15, SR16, SR17, SR18, SR19, SR200, SR245, SR251**
UCR28	**SR20, SR21, SR22, SR23, SR24, SR245**
UCR29	**SR245, SR261**
UCR30	**SR25, SR26, SR27, SR28, SR29, SR30**
UCR31	**SR251, SR252**
UCR32	**SR252, SR21, SR16**
UCR33	**SR252, SR48, SR49**
UCR34	**SR252, SR48, SR49**
UCR35	**SR31, SR32, SR33, SR36, SR244**
UCR36	**SR31, SR32, SR34, SR36, SR244**
UCR37	**SR31, SR32, SR35, SR36, SR244**
UCR38	**SR37, SR38, SR39, SR40, SR45, SR46, SR47, SR244**
UCR39	**SR37, SR38, SR39, SR40, SR244**
UCR40	**SR37, SR38, SR39, SR40, SR244**

UCR46	**SR248**
UCR47	**SR248, SR262, SR261**
UCR48	**SR248**
UCR49	**SR53, SR54**
UCR50	**SR55, SR56**
UCR51	**SR60**
UCR52	**SR60**
UCR53	**SR57**
UCR54	**SR58**
UCR55	**SR63, SR64, SR65, SR66, SR67, SR68, SR69, SR70, SR71, SR72, SR73, SR74, SR75, SR80, SR81, SR82, SR82, SR83, SR84, SR85, SR86, SR87, SR88, SR89 SR90, SR91, SR92**
UCR56	**SR63, SR64, SR65, SR66, SR67, SR68, SR69, SR70, SR71, SR72, SR73, SR74, SR80, SR81, SR82, SR82, SR83, SR84, SR85, SR86, SR87, SR88, SR89 SR75, SR90, SR91, SR92**
UCR57	**SR93, SR94, SR95, SR96, SR97, SR98, SR99, SR100, SR101, SR102, SR103, SR243**
UCR57a	**SR50, SR51, SR52, SR53, SR54, SR55, SR56, SR57, SR58, SR59, SR60, SR61, SR62**
UCR57b	**SR50, SR51, SR52, SR53, SR54, SR55, SR56, SR57, SR58, SR59, SR60, SR61, SR62**
UCR57c	**SR80, SR81, SR82, SR83, SR84, SR85, SR86, SR87, SR88, SR89, SR90, SR91, SR92**
UCR57d	**SR50, SR51, SR52, SR53, SR54, SR55, SR56, SR57, SR58, SR59, SR60, SR61, SR62**
UCR57e	**SR76, SR77, SR78, SR79**
UCR58	**SR241, SR242, SR254**
UCR59	**SR243**
UCR60	**SR93, SR94, SR95, SR96, SR97, SR98, SR99, SR100, SR101, SR102, SR103**
UCR61	**SR108, SR115**
UCR62	**SR108, SR115**
UCR63	**SR190, SR253**
UCR64	**SR253**
UCR65	**SR124, SR125, SR126, SR127, SR128, SR129, SR130, SR131, SR132, SR133, SR134, SR135, SR136, SR137, SR138, SR139, SR140, SR141, SR142, SR143**
UCR66	**SR254**
UCR67	**SR254, SR242, SR243**
UCR68	**SR241, SR243, SR254**
UCR69	**SR243**
UCR70	**SR115, SR116, SR117, SR118, SR119, SR120, SR121, SR122, SR123, SR111, SR112, SR113, SR114**
UCR71	**SR124, SR125, SR126, SR127, SR128, SR129, SR130, SR131, SR132, SR133, SR134, SR135, SR136, SR137, SR138, SR139, SR140, SR141, SR142, SR143**
UCR72	**SR124, SR130, SR131, SR132, SR133, SR134, SR247**
UCR73	**SR135, SR247**
UCR74	**SR247, SR135**
UCR75	**SR247, SR156, SR143, SR174**
UCR76	**SR247, SR152, SR139, SR170**
UCR77	**SR247, SR140, SR153, SR171**
UCR78	**SR247, SR141, SR154, SR172**
UCR79	**SR247**
UCR80	**SR247, SR132, SR133, SR134**

UCR81	**SR247, SR242, SR50, SR51, SR52, SR53, SR54, SR55, SR56, SR57, SR58, SR59,SR60, SR61, SR62**
UCR82	**SR247, SR241, SR254**
UCR83	**SR247, SR157, SR158, SR159, SR160, SR161**
UCR84	**SR247, SR127**
UCR86	**SR247, SR157, SR158, SR159, SR160, SR161**
UCR88	**SR247, SR157, SR158, SR159, SR160, SR161**
UCR89	**SR203, SR247**
UCR90	**SR247, SR162, SR163, SR164, SR160, SR159**
UCR91	**SR247, SR162, SR163, SR164, SR160, SR159, SR166, SR167**
UCR92	**SR144, SR145, SR146, SR147, SR148, SR149, SR150, SR151, SR152, SR153, SR154, SR155, SR156**
UCR93	**SR144, SR145, SR146, SR147, SR148, SR149, SR150, SR151, SR152, SR153, SR154, SR155, SR156**
UCR94	**SR157, SR158, SR159, SR160, SR161**
UCR95	**SR157, SR158, SR159, SR160, SR161**
UCR96	**SR247**
UCR97	**SR247, SR156, SR143, SR174**
UCR98	**SR247, SR168, SR150, SR137**
UCR99	**SR247, SR152, SR139, SR170**
UCR100	**SR247, SR138, SR151, SR169**
UCR101	**SR247, SR140, SR153, SR171**
UCR102	**SR247, SR141, SR154, SR172**
UCR103	**SR247, SR142, SR155, SR173**
UCR104	**SR247, SR142, SR155, SR173**
UCR105	**SR179**
UCR106	**SR264**
UCR108	**SR247, SR243 SR124, SR125, SR126, SR127, SR128, SR129, SR130, SR131, SR132, SR133, SR134, SR135, SR136, SR137, SR138, SR140, SR141, SR142,**
UCR109	**SR264, SR50, SR51, SR52, SR53, SR54, SR55, SR56, SR57, SR143, SR203 SR58, SR59, SR60, SR61, SR62**
UCR109a	**SR264, SR50, SR51, SR52, SR53, SR54, SR55, SR56, SR57, SR58, SR59, SR60, SR61, SR62, SR108, SR115**
UCR110	**SR192, SR193, SR194**
UCR111	**SR264, SR108, SR115, SR202**
UCR112	**SR189, SR190, SR191, SR218, SR219, SR222**
UCR115	**SR187**
UCR116	**SR180**
UCR117	**SR180**
UCR118	**SR180**
UCR119	**SR188**
UCR120	**SR181, SR192, SR193, SR194**

UCR121	**SR181, SR182, SR192, SR193, SR194, SR195, SR196, SR197**
UCR122	**SR180, SR192, SR193, SR194**
UCR123	**SR180, SR192, SR193, SR194**
UCR124	**SR179**
UCR125	**SR179, SR180**
UCR126	**SR192, SR193, SR194**
UCR127	**SR192, SR193, SR194**
UCR128	**SR181**
UCR129	**SR181, SR180**
UCR130	**SR186**
UCR131	**SR186**
UCR132	**SR129, SR135**
UCR133	**SR205, SR206, SR207**
UCR134	**SR209, SR198**
UCR135	**SR111, SR112, SR113, SR114, SR115, SR116, SR117, SR118 SR119, SR120, SR121, SR122, SR123**
UCR135a	**SR111, SR112, SR113, SR114, SR115, SR116, SR117, SR118 SR119, SR120, SR121, SR122, SR123**
UCR136	**SR205, SR206, SR207**
UCR137	**SR183, SR184, SR185, SR181**
UCR138	**SR212, SR213, SR214, SR215, SR216, SR217**
UCR139	**SR186**
UCR140	**SR186**
UCR141	**SR186**
UCR142	**SR186**
UCR143	**SR186**
UCR144	**SR208, SR257**
UCR145	**SR208**
UCR146	**SR208, SR251**
UCR147	**SR199, SR208, SR251**
UCR148	**SR208, SR42**
UCR149	**SR208**
UCR150	**SR208, SR50, SR51, SR52, SR53, SR54, SR55, SR56, SR57, SR58, SR59, SR60, SR61, SR62**
UCR151	**SR208, SR104, SR105, SR106, SR107, SR108, SR109, SR110, SR111, SR112 , SR113, SR114, SR115, SR116, SR117, SR118, SR119, SR120, SR121, SR122, SR123**
UCR152	**SR208, SR157, SR158, SR159, SR160, SR161**
UCR153	**SR208, SR155, SR142, SR173**
UCR154	**SR208, SR232**
UCR155	**SR208, SR256, SR63, SR64, SR65, SR66, SR67, SR68, SR69, SR70, SR71, SR72, SR73, SR74, SR75**
UCR156	**SR226**
UCR157	**SR227**
UCR158	**SR228**
UCR159	**SR229**
UCR160	**SR230**

UCR161	**SR231**
UCR162	**SR232**
UCR163	**SR233**
UCR164	**SR234**
UCR165	**SR235**
UCR166	**SR236**
UCR167	**SR237**
UCR168	**SR182, SR210, SR211, SR212, SR213, SR214, SR215**
UCR169	**SR258, SR259, SR260**
UCR170	**SR259, SR260**
UCR171	**SR261**
UCR172	**SR238**
UCR173	**SR281, SR282**
UCR174	**SR265, SR223, SR224**
UCR175	**SR266, SR223, SR224**
UCR176	**SR267,**
UCR177	**SR268, SR220, SR221**
UCR178	**SR269, SR220, SR221**
UCR179	**SR270, SR243**
UCR180	**SR271, SR243**
UCR181	**SR272**
UCR182	**SR273**
UCR183	**SR274**
UCR184	**SR275**
UCR185	**SR276, SR225**
UCR186	**SR277, SR225**
UCR187	**SR278, SR201**
UCR188	**SR279**
UCR189	**SR280**

4.2 Software requirements to user requirements

We now describe all user requirements related to a certain software requirement. Note that this is the transpose of the relation in the table of the previous section, which means that the user requirements and software requirements match.

SR1	**UCR24**
SR2	**UCR24**
SR3	**UCR24**
SR4	**UCR24**
SR5	**UCR24**
SR6	**UCR24**
SR7	**UCR24**
SR8	**UCR24**
SR9	**UCR24**
SR10	**UCR24**
SR11	**UCR24**
SR12	**UCR24**
SR13	**UCR24**
SR14	**UCR24**
SR15	**UCR27**
SR16	**UCR27**
SR17	**UCR27**
SR18	**UCR27**
SR19	**UCR27**
SR20	**UCR25, UCR28, UCR2D**
SR21	**UCR25, UCR28, UCR2D**
SR22	**UCR25, UCR28, UCR2D**
SR23	**UCR25, UCR28, UCR2D**
SR24	**UCR25, UCR28, UCR2D**
SR25	**UCR30**
SR26	**UCR30**
SR27	**UCR30**
SR28	**UCR30**
SR29	**UCR30**
SR30	**UCR30**
SR31	**UCR35, UCR36, UCR37**
SR32	**UCR35, UCR36, UCR37**
SR33	**UCR35**
SR34	**UCR36**
SR35	**UCR37**
SR36	**UCR35, UCR36, UCR37**
SR37	**UCR38, UCR39, UCR40**
SR38	**UCR38, UCR39, UCR40**
SR39	**UCR38, UCR39, UCR40**
SR40	**UCR38, UCR39, UCR40**

SR41	**UCR2a**
SR42	**UCR2a**
SR43	**UCR2a**
SR44	**UCR2a**
SR45	**UCR38**
SR46	**UCR38**
SR47	**UCR38**
SR48	**UCR2, UCR5, UCR33, UCR34**
SR49	**UCR2, UCR5, UCR33, UCR34**
SR50	**UCR57a, UCR57b, UCR57d, UCR81, UCR109, UCR109a, UCR150**
SR51	**UCR57a, UCR57b, UCR57d, UCR81, UCR109, UCR109a, UCR150**
SR52	**UCR57a, UCR57b, UCR57d, UCR81, UCR109, UCR109a, UCR150**
SR53	**UCR49, UCR57a, UCR57b, UCR57d, UCR81, UCR109, UCR109a, UCR150**
SR54	**UCR49, UCR57a, UCR57b, UCR57d, UCR81, UCR109, UCR109a, UCR150**
SR55	**UCR50, UCR57a, UCR57b, UCR57d, UCR81, UCR109, UCR109a, UCR150**
SR56	**UCR50, UCR57a, UCR57b, UCR57d, UCR81, UCR109, UCR109a, UCR150**
SR57	**UCR53, UCR57a, UCR57b, UCR57d, UCR81, UCR109, UCR109a, UCR150**
SR58	**UCR54, UCR57a, UCR57b, UCR57d, UCR81, UCR109, UCR109a, UCR150**
SR59	**UCR57a, UCR57b, UCR57d, UCR81, UCR109, UCR109a, UCR150**
SR60	**UCR51, UCR52, UCR57a, UCR57b, UCR57d, UCR81, UCR109, UCR109a, UCR150**
SR61	**UCR57a, UCR57b, UCR57d, UCR81, UCR109, UCR109a, UCR150**
SR62	**UCR57a, UCR57b, UCR57d, UCR81, UCR109, UCR109a, UCR150**
SR63	**UCR55, UCR56, UCR155**
SR64	**UCR55, UCR56, UCR155**
SR65	**UCR55, UCR56, UCR155**
SR66	**UCR55, UCR56, UCR155**
SR67	**UCR55, UCR56, UCR155**
SR68	**UCR55, UCR56, UCR155**
SR69	**UCR55, UCR56, UCR155**
SR70	**UCR55, UCR56, UCR155**
SR71	**UCR55, UCR56, UCR155**
SR72	**UCR55, UCR56, UCR155**
SR73	**UCR55, UCR56, UCR155**
SR74	**UCR55, UCR56, UCR155**
SR75	**UCR55, UCR56, UCR155**
SR76	**UCR57e**

SR77	**UCR57e**
SR78	**UCR57e**
SR79	**UCR57e**
SR80	**UCR55, UCR56, UCR57c**
SR81	**UCR55, UCR56, UCR57c**
SR82	**UCR55, UCR56, UCR57c**
SR83	**UCR55, UCR56, UCR57c**
SR84	**UCR55, UCR56, UCR57c**
SR85	**UCR55, UCR56, UCR57c**
SR86	**UCR55, UCR56, UCR57c**
SR87	**UCR55, UCR56, UCR57c**
SR88	**UCR55, UCR56, UCR57c**
SR89	**UCR55, UCR56, UCR57c**
SR90	**UCR55, UCR56, UCR57c**
SR91	**UCR55, UCR56, UCR57c**
SR92	**UCR55, UCR56, UCR57c**
SR93	**UCR57, UCR60**
SR94	**UCR57, UCR60**
SR95	**UCR57, UCR60**
SR96	**UCR57, UCR60**
SR97	**UCR57, UCR60**
SR98	**UCR57, UCR60**
SR99	**UCR57, UCR60**
SR100	**UCR57, UCR60**
SR101	**UCR57, UCR60**
SR102	**UCR57, UCR60**
SR103	**UCR57, UCR60**
SR104	**UCR151**
SR105	**UCR151**
SR106	**UCR151**
SR107	**UCR151**
SR108	**UCR61, UCR62, UCR109a, UCR111, UCR151**
SR109	**UCR151**
SR110	**UCR151**
SR111	**UCR70, UCR151, UCR135, UCR135a**
SR112	**UCR70, UCR151, UCR135, UCR135a**
SR113	**UCR70, UCR151, UCR135, UCR135a**
SR114	**UCR70, UCR151, UCR135, UCR135a**
SR115	**UCR61, UCR62, UCR70, UCR109a, UCR111, UCR151, UCR135, UCR135a**
SR116	**UCR70, UCR151, UCR135, UCR135a**

SR117	**UCR70, UCR151, UCR135, UCR135a**
SR118	**UCR70, UCR151, UCR135, UCR135a**
SR119	**UCR70, UCR151, UCR135, UCR135a**
SR120	**UCR70, UCR151, UCR135, UCR135a**
SR121	**UCR70, UCR151, UCR135, UCR135a**
SR122	**UCR70, UCR151, UCR135, UCR135a**
SR123	**UCR70, UCR151, UCR135, UCR135a**
SR124	**UCR65, UCR71, UCR72, UCR108**
SR125	**UCR65, UCR71, UCR108**
SR126	**UCR65, UCR71, UCR108**
SR127	**UCR65, UCR71, UCR84, UCR108**
SR128	**UCR65, UCR71, UCR108**
SR129	**UCR65, UCR71, UCR108, UCR132**
SR130	**UCR65, UCR71, UCR72, UCR108**
SR131	**UCR65, UCR71, UCR72, UCR108**
SR132	**UCR65, UCR71, UCR72, UCR80 UCR108**
SR133	**UCR65, UCR71, UCR72, UCR80 UCR108**
SR134	**UCR65, UCR71, UCR72, UCR80 UCR108**
SR135	**UCR65, UCR71, UCR73, UCR74, UCR108, UCR132**
SR136	**UCR65, UCR71, UCR108**
SR137	**UCR65, UCR71, UCR98, UCR108**
SR138	**UCR65, UCR71, UCR100, UCR108**
SR139	**UCR65, UCR71, UCR76, UCR99**
SR140	**UCR65, UCR71, UCR77, UCR101, UCR108**
SR141	**UCR65, UCR71, UCR78, UCR102 UCR108**
SR142	**UCR65, UCR71, UCR103, UCR104, UCR108**
SR143	**UCR65, UCR71, UCR75, UCR97, UCR108**
SR144	**UCR92, UCR93**
SR145	**UCR92, UCR93**
SR146	**UCR92, UCR93**
SR147	**UCR92, UCR93**
SR148	**UCR92, UCR93**
SR149	**UCR92, UCR93**
SR150	**UCR92, UCR93, UCR98**
SR151	**UCR92, UCR93, UCR100**
SR152	**UCR76, UCR92, UCR93, UCR99**
SR153	**UCR77, UCR92, UCR93, UCR101**
SR154	**UCR78, UCR92, UCR93, UCR102**
SR155	**UCR92, UCR93, UCR103, UCR104, UCR153**
SR156	**UCR75, UCR92, UCR93, UCR97**

SR157	**UCR83, UCR86, UCR88, UCR94, UCR95, UCR152**
SR158	**UCR83, UCR86, UCR88, UCR94, UCR95, UCR152**
SR159	**UCR83, UCR86, UCR88, UCR90, UCR91 UCR94, UCR95, UCR152**
SR160	**UCR83, UCR86, UCR88, UCR90, UCR91 UCR94, UCR95, UCR152**
SR161	**UCR83, UCR86, UCR88, UCR94, UCR95, UCR152**
SR162	**UCR14, UCR15, UCR90, UCR91**
SR163	**UCR14, UCR15, UCR90, UCR91**
SR164	**UCR14, UCR15, UCR90, UCR91**
SR165	**UCR14, UCR15**
SR166	**UCR91**
SR167	**UCR91**
SR168	**UCR98**
SR169	**UCR100**
SR170	**UCR76, UCR99**
SR171	**UCR77, UCR101**
SR172	**UCR78, UCR102**
SR173	**UCR103, UCR104, UCR153**
SR174	**UCR75, UCR97**
SR175	**UCR94, UCR96, UCR100, UCR103, UCR104**
SR176	**UCR16**
SR177	**UCR16**
SR178	**UCR16**
SR179	**UCR105, UCR124, UCR125**
SR180	**UCR116, UCR117, UCR118, UCR122, UCR123, UCR125, UCR129**
SR181	**UCR120, UCR121, UCR128, UCR129, UCR137**
SR182	**UCR121, UCR168**
SR183	**UCR137**
SR184	**UCR137**
SR185	**UCR137**
SR186	**UCR130, UCR131, UCR139, UCR140, UCR141, UCR142, UCR143**
SR187	**UCR115**
SR188	**UCR119**
SR189	**UCR112**
SR190	**UCR63, UCR112**
SR191	**UCR112**
SR192	**UCR110, UCR120, UCR121, UCR122, UCR123, UCR126, UCR127**
SR193	**UCR110, UCR120, UCR121, UCR122, UCR123, UCR126, UCR127**
SR194	**UCR110, UCR120, UCR121, UCR122, UCR123, UCR126, UCR127**
SR195	**UCR121**
SR196	**UCR121**

SR197	**UCR121**
SR198	**UCR134**
SR199	**UCR147**
SR200	**UCR27**
SR201	**UCR187**
SR202	**UCR111**
SR203	**UCR89**
SR204	**UCR108**
SR205	**UCR133, UCR136**
SR206	**UCR133, UCR136**
SR207	**UCR133, UCR136**
SR208	**UCR144, UCR145, UCR146, UCR147, UCR148, UCR149, UCR150, UCR151, UCR152, UCR153, UCR154, UCR155**
SR209	**UCR134**
SR210	**UCR168**
SR211	**UCR168**
SR212	**UCR138, UCR168**
SR213	**UCR138, UCR168**
SR214	**UCR138, UCR168**
SR215	**UCR138, UCR168**
SR216	**UCR138**
SR217	**UCR138**
SR218	**UCR112**
SR219	**UCR112**
SR220	**UCR177, UCR178**
SR221	**UCR177 , UCR178**
SR222	**UCR222**
SR223	**UCR174, UCR175**
SR224	**UCR174, UCR175**
SR225	**UCR185, UCR186**
SR226	**UCR156**
SR227	**UCR157**
SR228	**UCR19, UCR158**
SR229	**UCR19, UCR159**
SR230	**UCR160**
SR231	**UCR161**
SR232	**UCR154, UCR162**
SR233	**UCR163**
SR234	**UCR164**
SR235	**UCR165**
SR236	**UCR166**

SR237	**UCR167**
SR238	**UCR3, UCR4, UCR7, UCR10, UCR11, UCR172**
SR239	**UCR26**
SR240	**UCR8, UCR9, UCR10, UCR11**
SR241	**UCR13, UCR14, UCR15, UCR16, UCR17, UCR18, UCR19, UCR20 UCR21, UCR22, UCR23, UCR58, UCR68, UCR82**
SR242	**UCR58, UCR67, UCR81**
SR243	**UCR57, UCR59, UCR67, UCR68, UCR69, UCR108, UCR179, UCR180**
SR244	**UCR13, UCR35, UCR36, UCR37, UCR38, UCR39, UCR40**
SR245	**UCR2b, UCR21, UCR27, UCR28, UCR29**
SR246	**UCR2c, UCR22, UCR24, UCR25, UCR26**
SR247	**UCR72, UCR73, UCR74, UCR75, UCR76, UCR77, UCR78, UCR79, UCR80 UCR81, UCR82, UCR83, UCR84, UCR85, UCR86, UCR87, UCR88, UCR89, UCR90 UCR91, UCR96, UCR97, UCR98, UCR99, UCR100, UCR101, UCR102, UCR103 UCR104, UCR108**
SR248	**UCR18, UCR23, UCR46, UCR47, UCR48**
SR249	**UCR15, UCR16**
SR250	**UCR2, UCR2b, UCR2c, UCR2d**
SR251	**UCR5, UCR17, UCR20, UCR27, UCR31, UCR146, UCR147**
SR252	**UCR20, UCR31, UCR32, UCR33, UCR34**
SR253	**UCR6, UCR63, UCR64**
SR254	**UCR58, UCR66, UCR67, UCR68, UCR82**
SR255	**UCR1**
SR256	**UCR155**
SR257	**UCR144**
SR258	**UCR169**
SR259	**UCR169, UCR170**
SR260	**UCR169, UCR170**
SR261	**UCR29, UCR47, UCR171**
SR262	**UCR1, UCR47**
SR263	**UCR12**
SR264	**UCR106, UCR109, UCR109a, UCR111**
SR265	**UCR174**
SR266	**UCR175**
SR267	**UCR176**
SR268	**UCR177**
SR269	**UCR178**
SR270	**UCR179**
SR271	**UCR180**
SR272	**UCR181**
SR273	**UCR182**
SR274	**UCR183**
SR275	**UCR184**
SR276	**UCR185**
SR277	**UCR186**
SR278	**UCR187**
SR279	**UCR188**
SR280	**UCR189**
SR281	**UCR173**
SR282	**UCR173**

Appendix A
User Interface

This appendix includes mock-ups that show the design decisions that we made for this system.

A.1 User application

Note that the pictures are from a browser version of the application. The final version may be different from the mock-ups shown below.

A.1.1 Splash screen

When booting up the application, the splash screen shown below is displayed before the application is loaded. It features the logo of the TravelMatch app.

Figure A.1: Splash screen

A.1.2 Start screen

After the splash screen is shown, this screen is shown to show the user the options they can make to continue with the app. The user can choose to be a guest user, register with an e-mail address, or log in with Facebook or with an e-mail address.

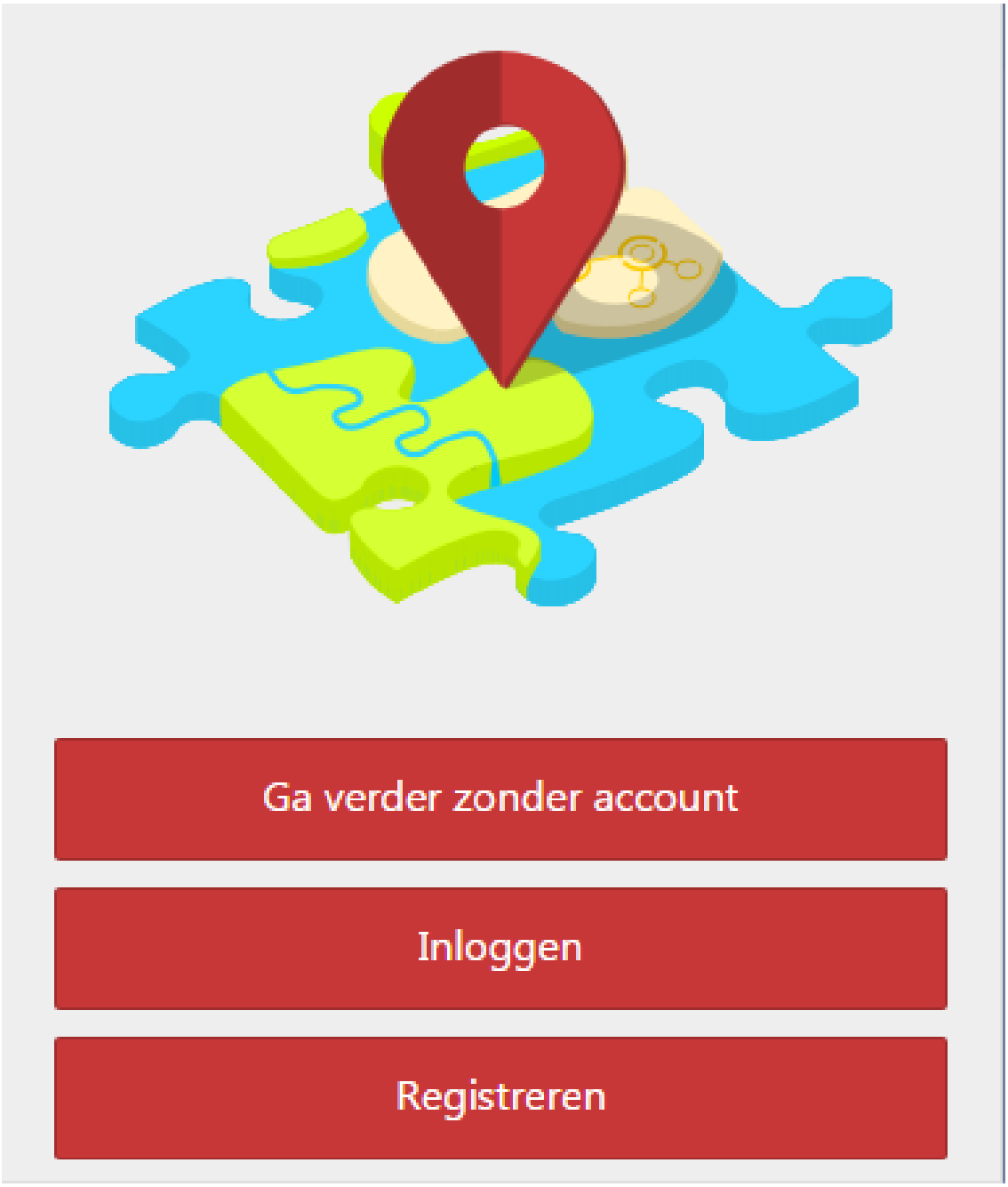

Figure A.2: Startscreen

A.1.3 Vacation details

When the the user has logged in as guest or has authenticated, the vacation details screen is shown. In this screen, the user can fill in data needed to find any destination fitting for their vacation. When the user clicks on the hamburger icon in the top right, they can go to a sidebar menu. If the user enters the vacation details and starts the advice, they are redirected to the interest analysis. At the top, an instruction is given about the TravelMatch app to describe the actions needed to get a recommendation. It has been left out for space constraints.

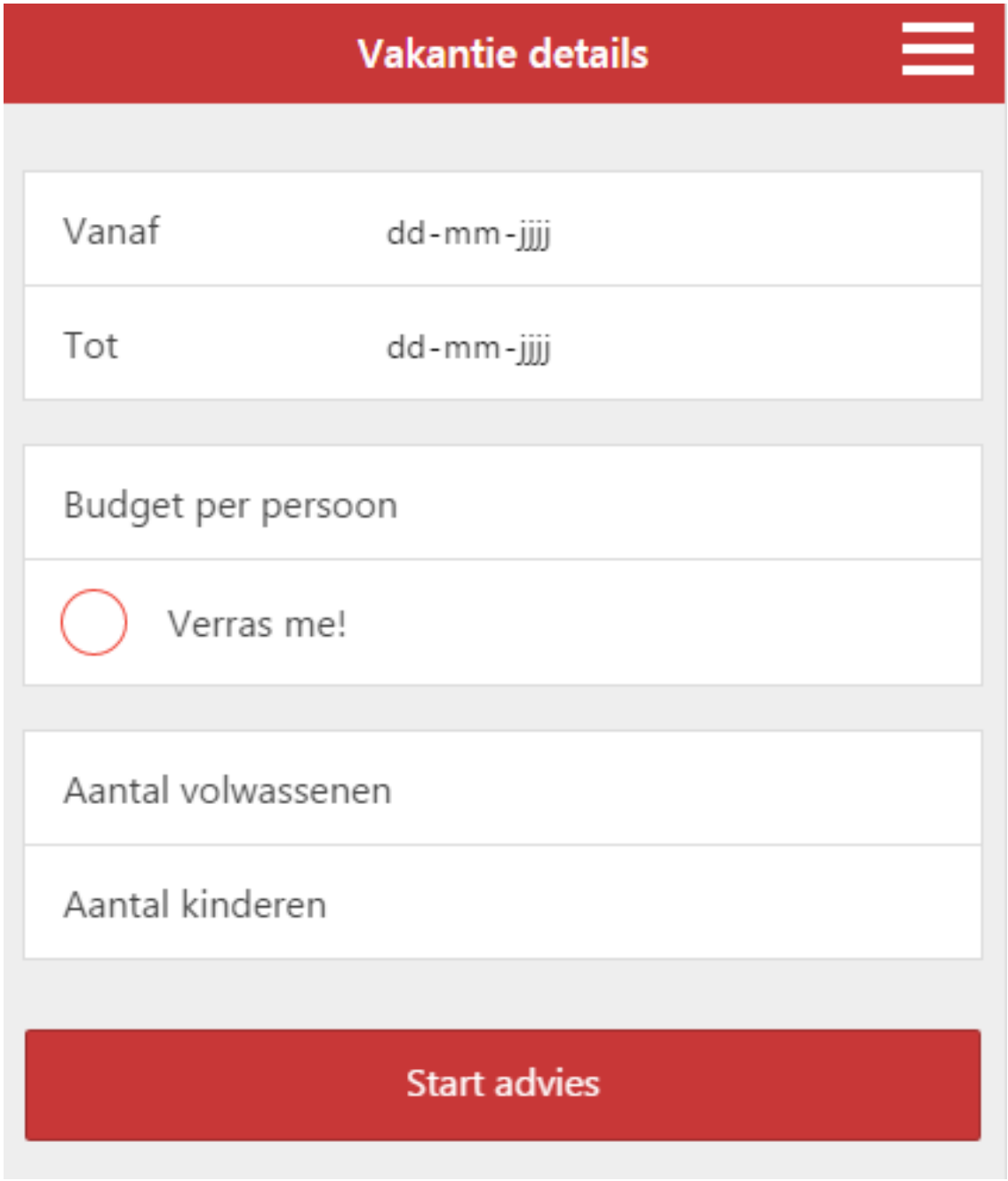

Figure A.3: Vacation detail input screen

A.1.4 Sidebar

The sidebar menu shown below partially overlays the current screen, and shows the different screen the user can view by clicking. Furthermore, it has the functionality to log out or log in. In this case, the log out button is shown, but otherwise, log in and register buttons are shown. When the sidebar is shown, the hamburger button is replaced by a cross icon, to indicate that the user can close the sidebar by using this button. In this sidebar the user can view all available services for them in the current stage of obtaining a recommendation. As such, after the user has judged a certain amount of images the sidebar will display different links.

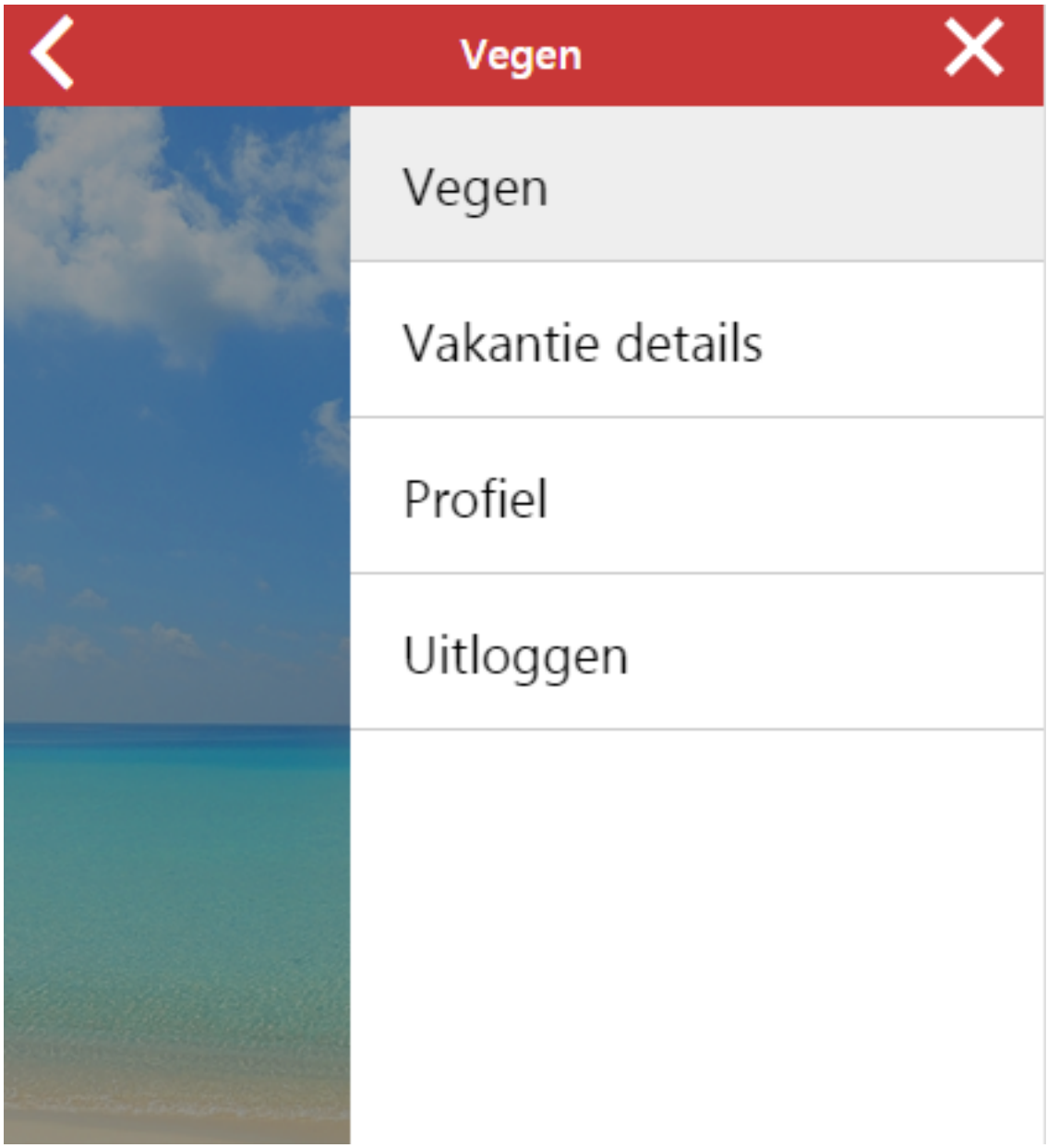

Figure A.4: Sidebar

A.1.5 Registration

When the user is on the registration page, they can fill in their e-mail address and their password (which must be entered twice). When the submit button is pressed, the data is sent to the back end. The user will receive feedback on whether they have registered correctly, the passwords did not match, or the e-mail address is incorrect.

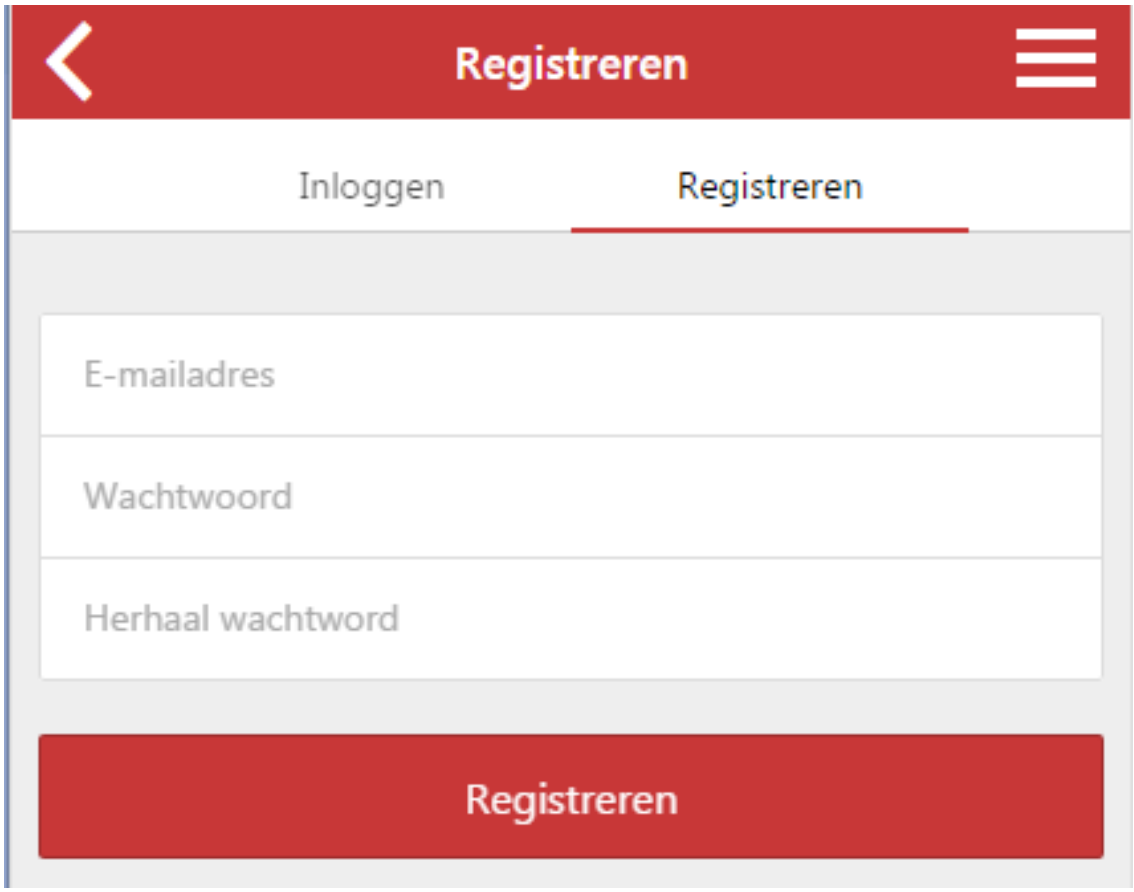

Figure A.5: Registering an account

A.1.6 Registration error

When the user is in any screen in the app, they can receive a pop-up which can be dismissed by clicking the red button at the bottom of the pop-up. The pop-up explains the an error that happened or gives feedback on actions made by the user. In the picture below the passwords did not match and so the passwords field have been cleared. The user can try again to fill in their passwords correctly.

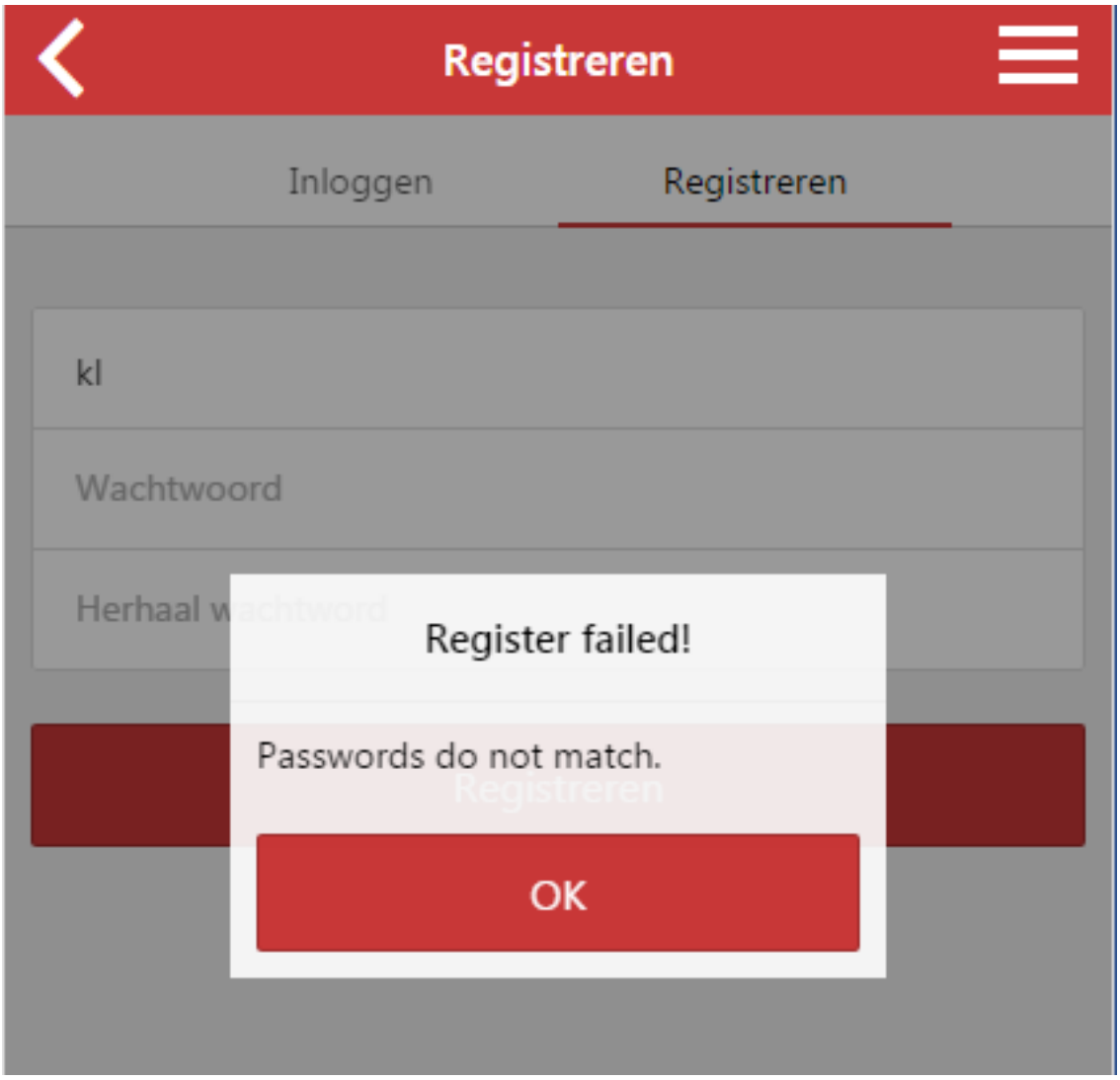

Figure A.6: Registering an account failed

A.1.7 Logging in

When the user goes to the log in screen, they are asked to input their e-mail address and password, or to click the "Login with Facebook" button. Pushing either button generates feedback on what was done in the back end, or what was done wrong by the user. Logging in with Facebook for the first time will also prompt a login for Facebook, and afterwards an authorization for the TravelMatch application. If the user has a Facebook application installed on their device, this application is started instead in order to authenticate the user more easily.

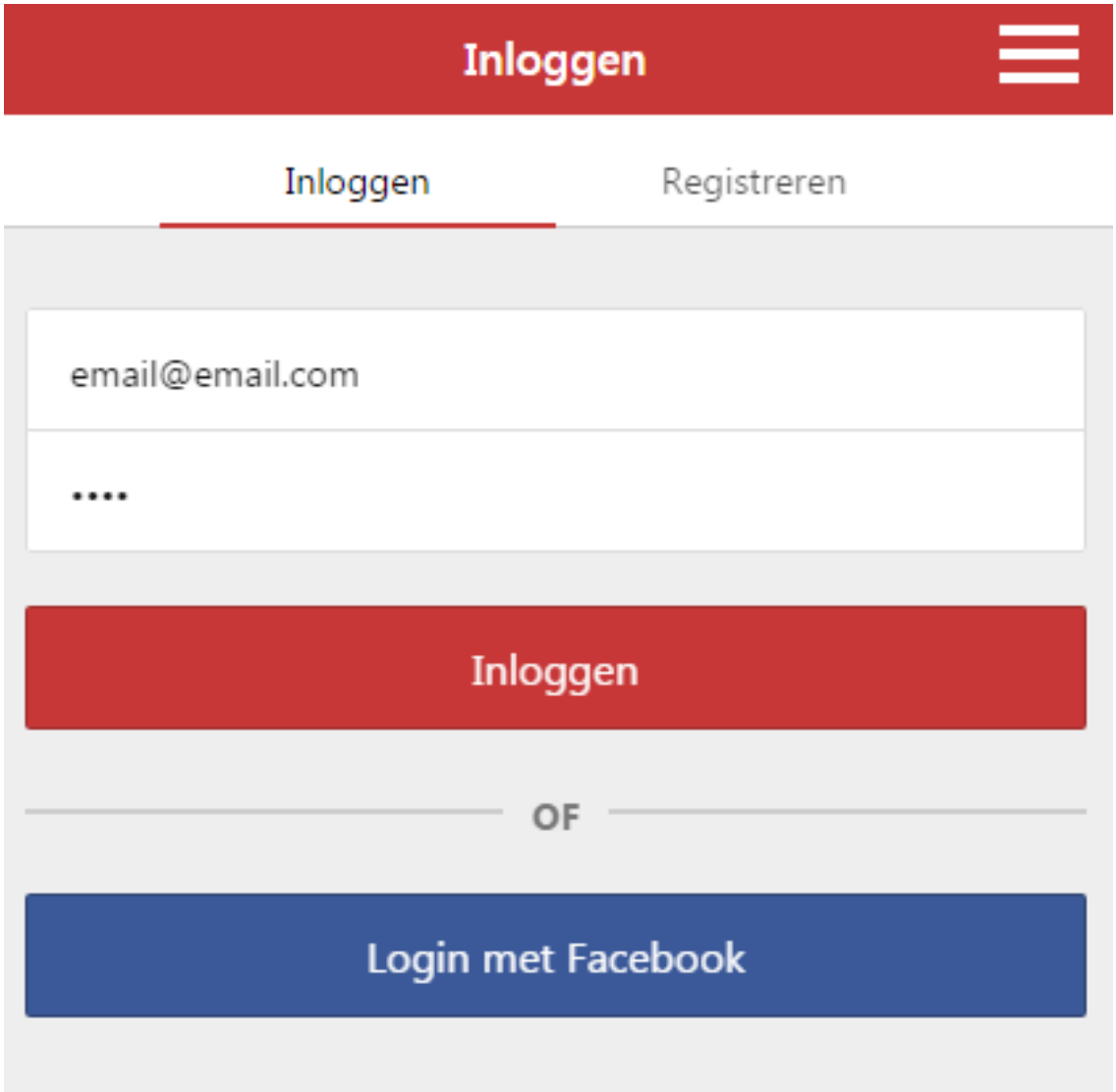

Figure A.7: Login Screen

A.1.8 Profile

When the user is logged in, they can go to the profile page which stores certain data about them. At the moment, name, gender and birthday are stored, which are shown every time on this page and updated every time the button is pressed. In the case the user logs out and in a later stage logs in again, they can retrieve the previously filled in data by going to this screen.

Figure A.8: Profile

A.1.9 Interest analysis example

In the interest analysis screen, the user is shown an image from the database with two buttons for "like" and "dislike". In this page the user can drag the image right and left, which corresponds respectively with a "like" or a "dislike" of the diplayed image below. When dragging, underneath the image the next image can be seen. The purpose of liking or disliking the image is create a Travel DNA in the back end, which is used by the AI to generate a vacation recommendation. In the bottom a progress bar can be seen, where blue indicates the progress. Once the blue bar reaches the right edge of the screen, the interest analysis is finished.

Figure A.9: Picture to be liked or disliked

A.1.10 Loading screen

A picture to fill the void when there are no pictures loaded in the cache of the app. The user should be connected to the internet in order to use the application extensively, so this screen should be shown as little as possible. A placeholder loading screen is depicted below.

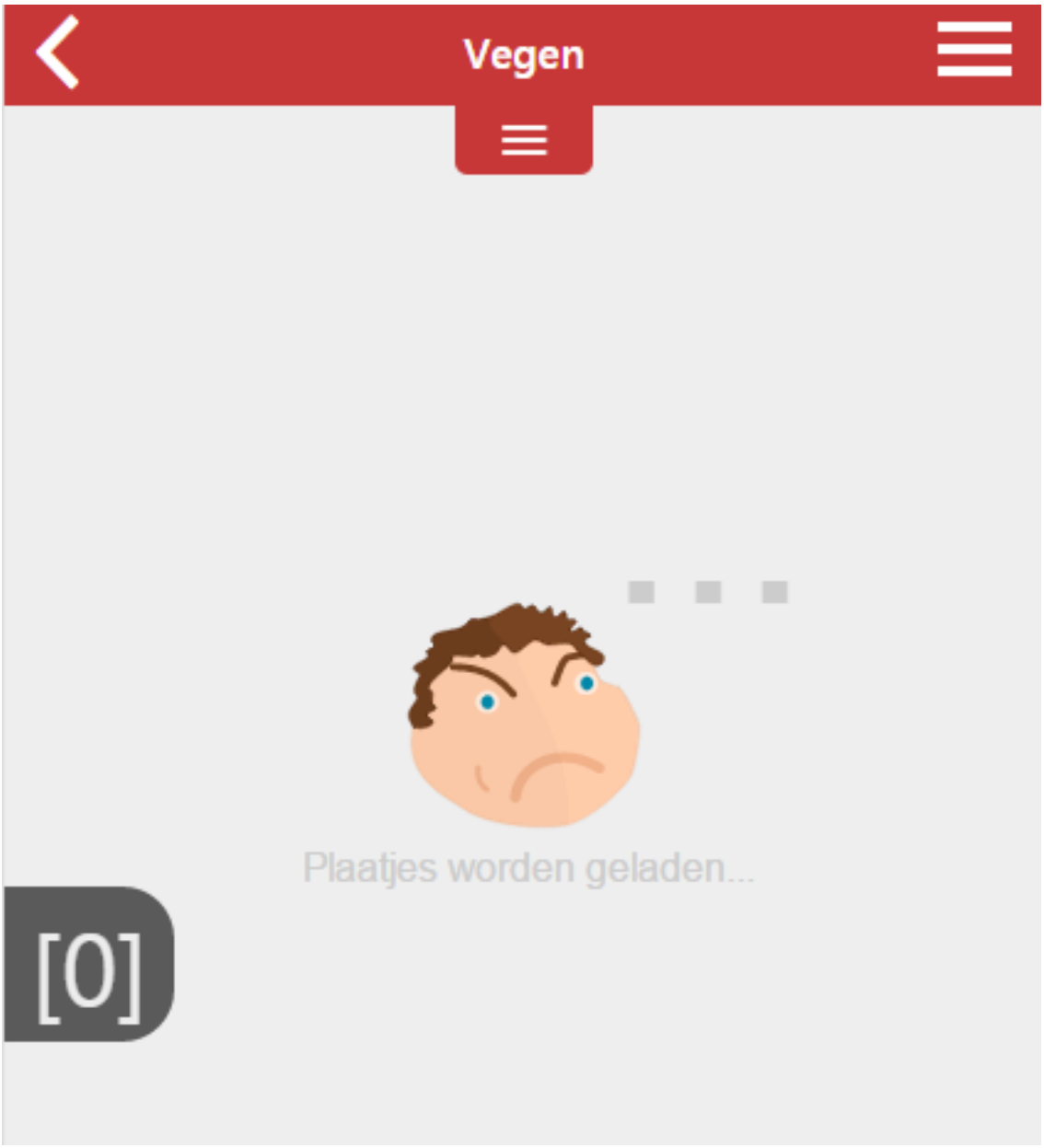

Figure A.10: Load screen if there are no images left in the buffer

A.1.11 Hotel overview screen

If the user has swiped a set amount of images, a recommendation is shown, which happens on this screen. The hotel overview screen shows the trips that are preferred within the recommended destination. The user can click on the second advice button to get the second recommended destination, and toggle between the two.

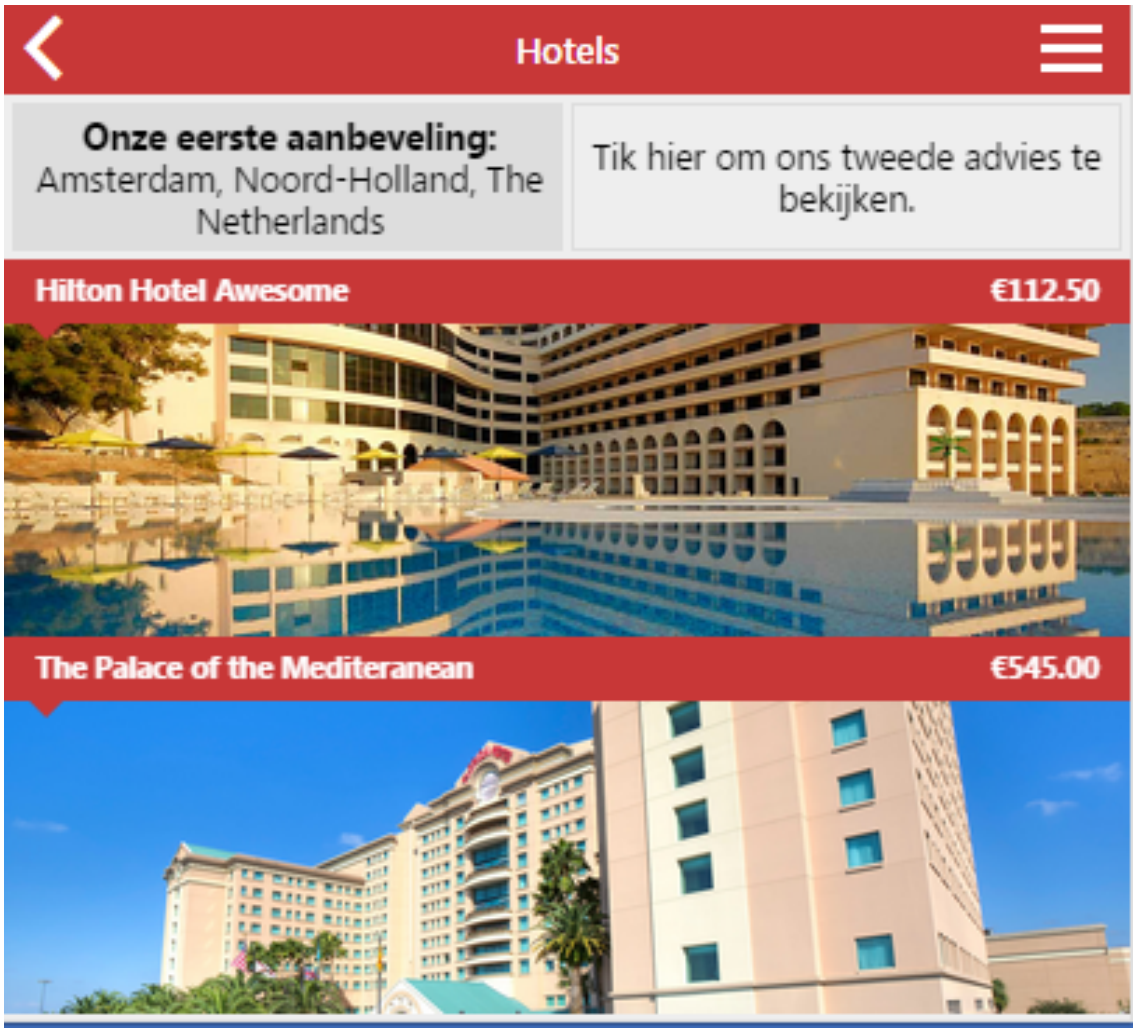

Figure A.11: An overview of the trips recommended

A.1.12 Hotel details

In the hotel overview screen, when the user clicks on the image of an hotel, the hotel description is shown and a button is displayed to book the trip.

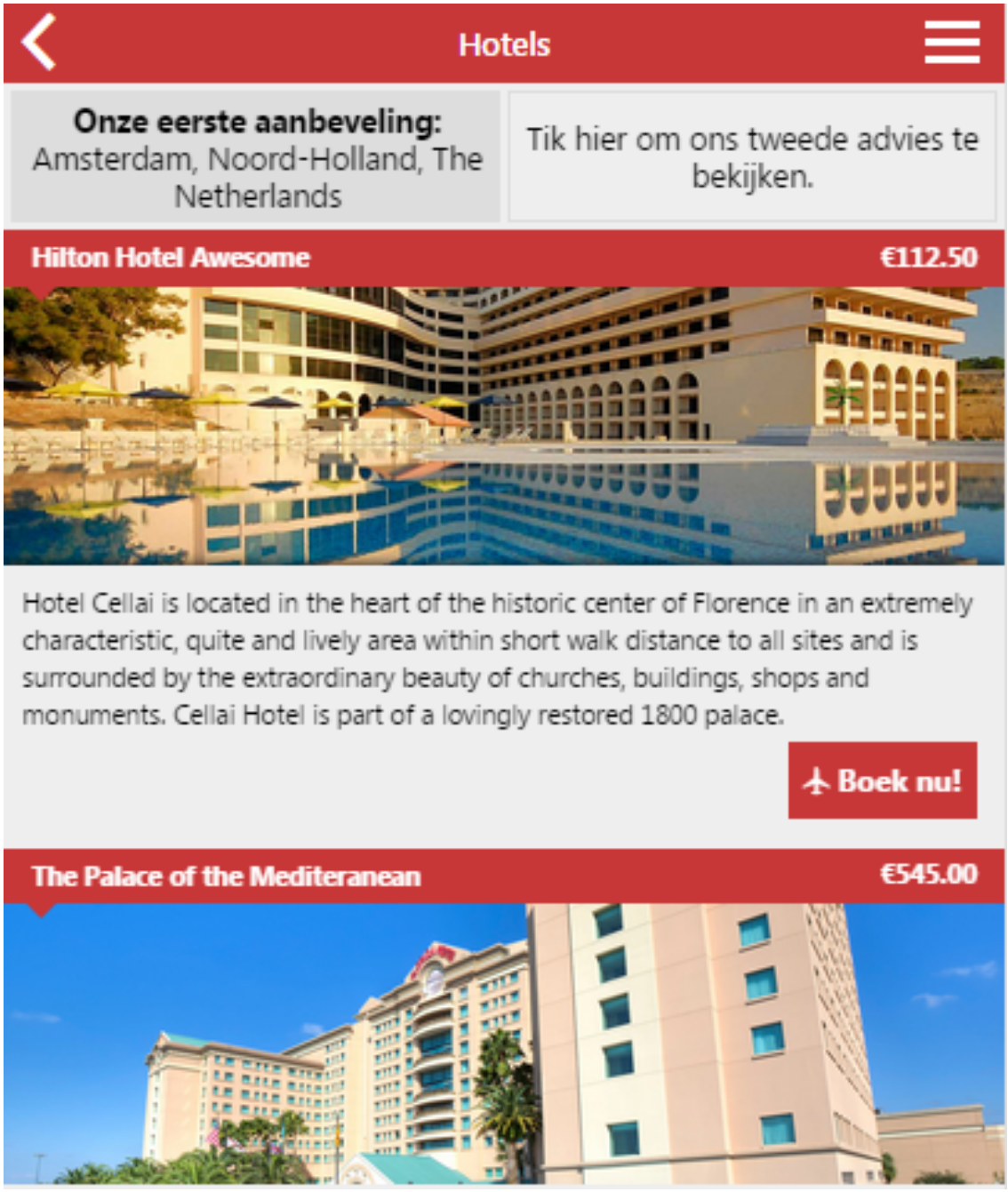
Hotels
Onze eerste aanbeveling:
Amsterdam, Noord-Holland, The Netherlands
Tik hier om ons tweede advies te bekijken.
Hilton Hotel Awesome
€112.50
Hotel Cellai is located in the heart of the historic center of Florence in an extremely characteristic, quite and lively area within short walk distance to all sites and is surrounded by the extraordinary beauty of churches, buildings, shops and monuments. Cellai Hotel is part of a lovingly restored 1800 palace.
Boek nu!
The Palace of the Mediteranean
€545.00

Figure A.12: a part of the hotel overview screen to display more details

A.2 CMS

These mock-ups are from an incomplete version of the CMS; the final version can be different from the mock-ups shown below. In the mock-ups below, the Google Chrome browser is used to show the CMS.

A.2.1 Login

The login page for the CMS, when the fields for username and passwords are filled in correctly, the user is authenticated and directed to the overview of the CMS.

Figure A.13: CMS login screen

A.2.2 Overview

An overview page for all the available changeable data at the moment in the database. In the future iLysian will employ personnel to manage the data stored in the database via this CMS. The look of the overview is dependent on the rights that the user has.

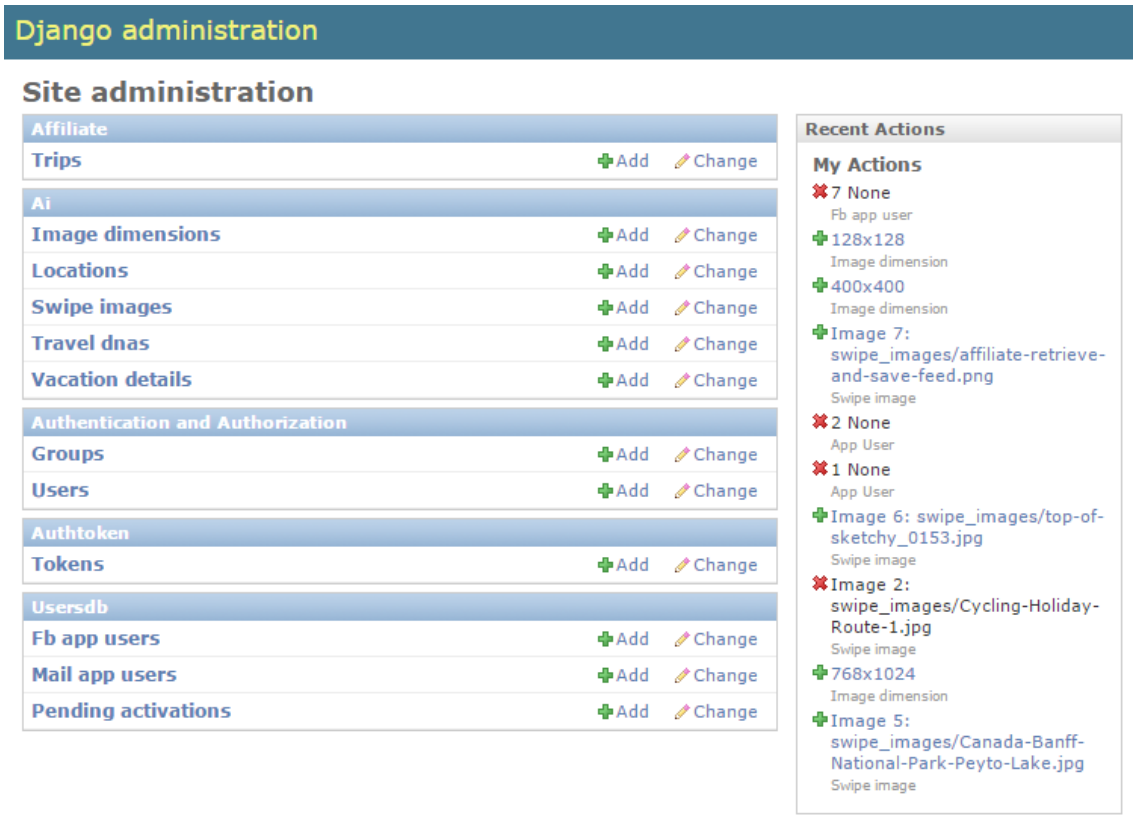

Figure A.14: CMS overview screen of the functionality

A.2.3 Adding images

A page for adding images to the database. The user who added the image must be given in order to be able to trace back who did what later. The "activated" check box facilitates direct deployment in the TravelMatch.

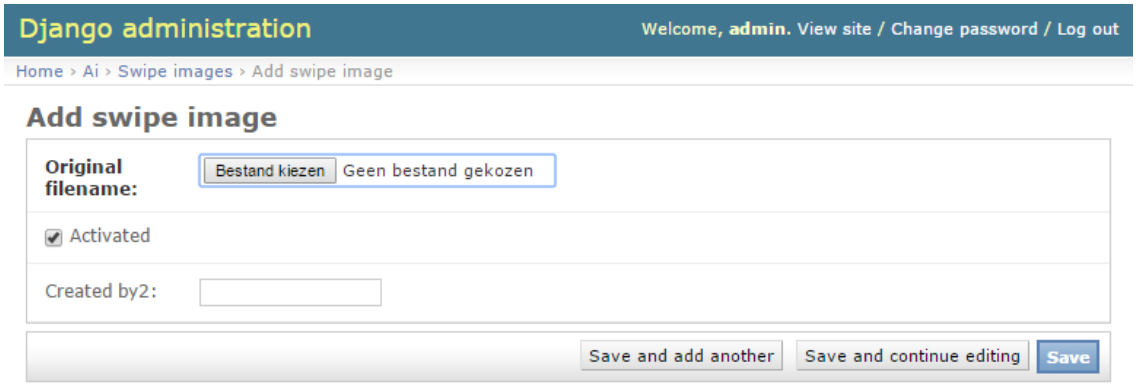

Figure A.15: Picture input screen

A.2.4 Images overview

An overview of all the added pictures. A thumbnail is shown to easily identify the stored pictures.

Figure A.16: Overview of all the pictures added in the CMS

A.2.5 User overview

An overview of all users who registered within the application. Users can be easily identified by their e-mail or name, and are ordered by user ID.

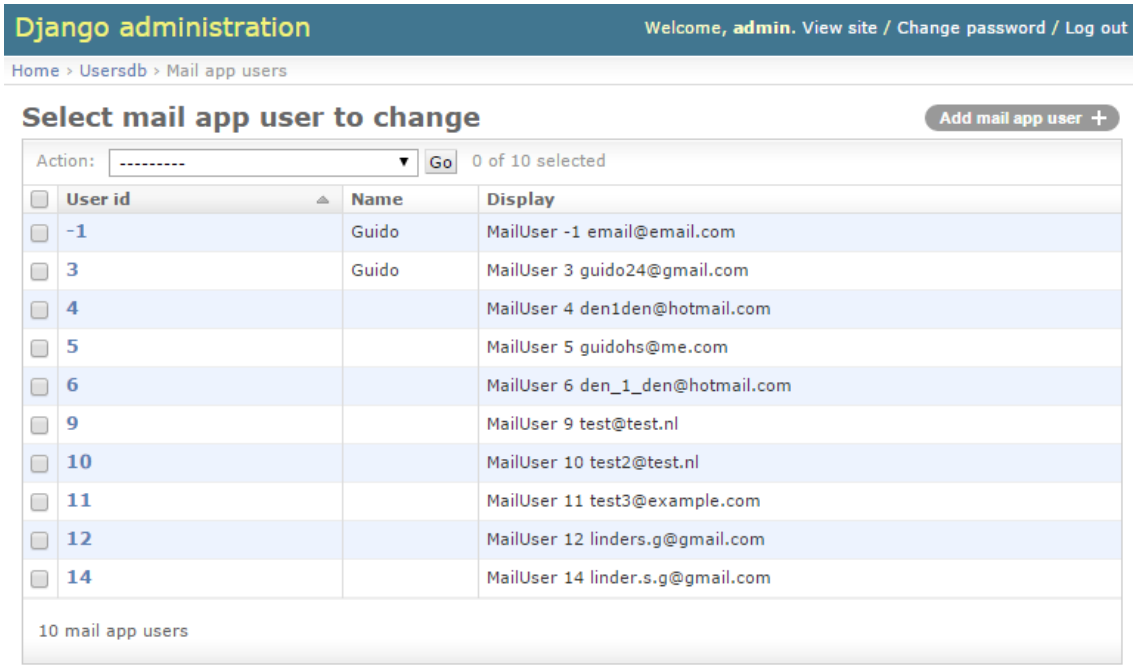
Django administration
Welcome, admin. View site / Change password / Log out
Home › Usersdb › Mail app users
Select mail app user to change
Add mail app user +
Action: --------- Go 0 of 10 selected
User id
Name
Display
-1 Guido MailUser -1 email@email.com
3 Guido MailUser 3 guido24@gmail.com
4 MailUser 4 den1den@hotmail.com
5 MailUser 5 guidohs@me.com
6 MailUser 6 den_1_den@hotmail.com
9 MailUser 9 test@test.nl
10 MailUser 10 test2@test.nl
11 MailUser 11 test3@example.com
12 MailUser 12 linders.g@gmail.com
14 MailUser 14 linder.s.g@gmail.com
10 mail app users

Figure A.17: Overview of the users

A.2.6 Single user

An example of the view of a single user entry in the database. This is an overview of all the input fields that are enabled to be shown by Django.

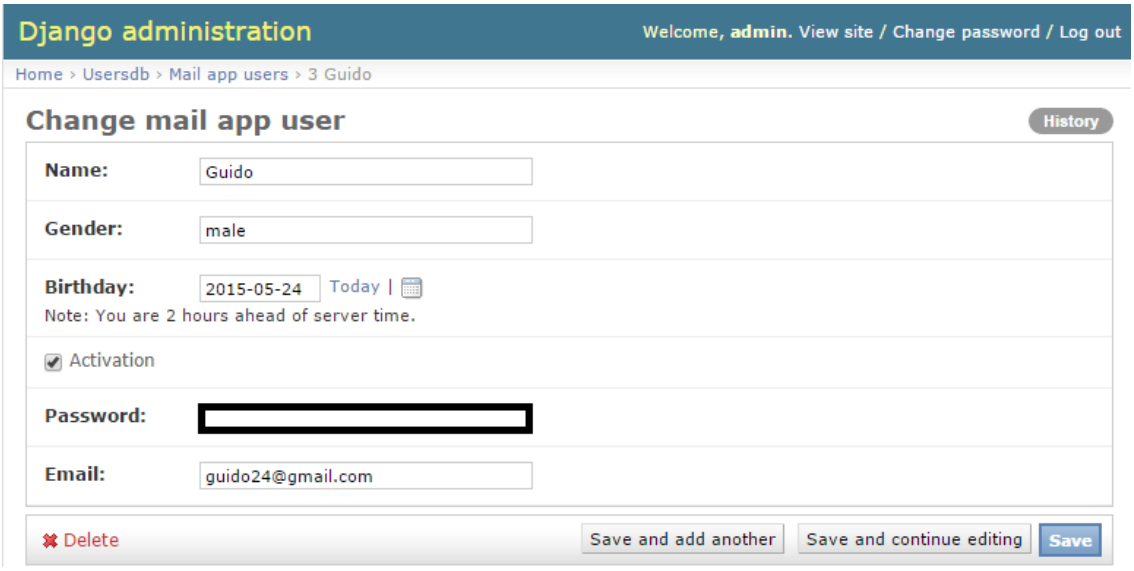

Figure A.18: An example of a single user